Set Intervals in Constraint Logic Programming: Definition and implementation of a language

Carmen Gervet

N^o d'ordre : **469**

THESE

présentée devant

L'Université de Franche-Comté

U.F.R. des Sciences et Techniques

pour obtenir le titre de

Docteur de l'Université de Franche-Comté
mention Informatique

Doctor Communitatis Europeae

Set Intervals in Constraint Logic Programming:

Definition and implementation of a language

par

Carmen Gervet

soutenue le 15 septembre 1995 devant la commission d'examen

MM.	**Yves**	**Deville**	Rapporteurs
	Hervé	**Gallaire**	
	Ugo	**Montanari**	
Mr.	**Bruno**	**Legeard**	Directeur
MM.	**Pierre**	**Baptiste**	Examinateurs
	Jean-Jacques	**Chabrier**	
	Alexander	**Herold**	
	Mark	**Wallace**	

A mes parents,
mille mercis.

Remerciements

Ce travail a bénéficié de nombreuses contributions mais il y a deux personnes sans lesquelles ce qui suit n'aurait pas vu le jour: Jean-Yves Cras et Mark Wallace.

Jean-Yves Cras qui, autour d'un grand café noir, m'a proposé ce sujet de thèse. Sa fougue et sa foi en l'avenir de la programmation par contraintes ont suffi pour me convaincre du devenir de ce projet. Qu'il en soit grandement remercié.

Mark Wallace m'a invitée à intégrer l'équipe CORE (COnstraint REasoning) à l'ECRC pour réaliser ce projet; son enthousiasme, son aide dans la mise en oeuvre d'idées nouvelles et sa joie de vivre, ont fait de ces années de thèse une aventure heureuse. Ses commentaires m'ont été d'une grande aide pour la version finale de cette thèse. Qu'il en soit grandement remercié.

A Bruno Legeard, qui en tant que directeur de recherche a assuré le bon déroulement de ces trois années de thèse, en m'accordant sa confiance et ses nombreux conseils tant sur les plans scientifique que pratique, merci.

A Rémi Lescoeur et toute l'équipe Charme qui m'ont chaleureusement accueillie pendant un an au centre d'intelligence artificielle de Bull (CEDIAG) et m'ont initiée à la programmation par contraintes avant que je ne gagne l'ECRC, merci.

A Alexander Herold et toute l'équipe ECL^iPS^e *à l'ECRC, qui ont fait part d'un soutien actif en m'intégrant dans une bulle anglophone au milieu d'un monde germanico-bavarois, merci.*

A Gérard Comyn et Alessandro Giacalone qui m'ont donné de travailler à l'ECRC, un lieu de recherche idéal pour faire une thèse, merci.

A Pascal Brisset, Micha Meier et Joachim Schimpf, qui m'ont appris à développer proprement un langage de programmation; pour leur aide technique d'une grande richesse et leur disponibilité de chaque instant, merci.

A Yves Deville, Hervé Gallaire et Ugo Montanari qui ont consacré une part de leur temps précieux pour se plonger dans mes travaux et évaluer cette thèse, merci.

Aux examinateurs, Pierre Baptiste, Jean-Jacques Chabrier, Alexander Herold, Bruno Legeard, Mark Wallace, qui ont accepté d'être membres du jury, merci.

Une partie de ce document est basée sur un papier publié. A Yves Deville, Pascal Van Hentenryck, Joxan Jaffar et les correcteurs anonymes, qui ont lu ce papier et dont les commentaires ont été d'une grande valeur, merci.

A tous ceux qui ont contribué à la réalisation de ce document par leurs conseils, commentaires et relectures: Jean-Marc Andreoli, Frédéric Benhamou, Stéphane Bressan, Pascal Brisset, Alexander Herold, Gabriel Kuper, Bruno Legeard, Mark Wallace, et tout particulièrement Gabriel Kuper et Alexander Herold qui ont lu et relu les versions préliminaires de cette thèse en portant un jugement critique et constructif tant sur le fond que sur la forme de mon fran-glais, merci.

Enfin, pour leur soutien sans mesure et leur profonde amitié, Armelle, François, Axel-Frank, Bob, Thom et Andréa, merci à vous.

Abstract

We propose a formal and practical framework for defining and implementing a new constraint logic programming language over sets called Conjunto. The main motivation for this work was to overcome current problems in solving set-based combinatorial search problems. A set in Conjunto is constrained to range over a so called set domain specified as a set interval. Until recently, most work on embedding sets in logic programming and constraint logic programming has focused on set constructors and complete solvers. These approaches aimed at exploiting the expressiveness of sets. Our study of these languages came to the conclusion that complete solvers have severe efficiency problems due to the nondeterministic nature of set unification.

Set-based search problems are modelled in the language as set domain constraint satisfaction problems, in which the nodes are variables ranging over a set domain, and the arcs are set constraints. In addition it provides a set of additional constraints, namely the graduated constraints, which define a relation between sets and integers or integer domains. We are thus able to deal with set-based optimization problems which apply cost functions to quantifiable, *i.e.* arithmetic, terms, while working on sets.

Conjunto has been implemented using a constraint logic programming platform. The constraint solver is based on consistency techniques: a set of transformation rules perform interval reasoning over the domain bounds to infer local consistency. For efficiency reasons, completeness of the solver has been given up. The solver is described as a transition system checking one constraint at a time.

The formal framework describes how elements from the computation domain, *i.e.* the class of definable sets, can be approximated by elements from the constraint domain (set intervals), over which the computations are performed. It describes the approximations and closure operations guaranteeing that any computable solution lies in the approximations.

The practical viability of the language is demonstrated by a set of applications from operations research and combinatorial mathematics. They show the ability of the language to model set based problems in a natural and concise way while keeping the solving process efficient. Therefore they show that the trade-off between expressiveness and efficiency proposed in this thesis leads to a practical system.

Contents

List of Figures

Introduction

Ne corrige pas le mauvais,
mais augmente le bon.[1]

[1] All the citations given in this document are from "Dialogues avec l'ange", a document taken down by Gitta Mallasz.

Motivation

Once upon a time there was a big universe called a "universal set", formed by the union of subparts or subsets. The least element of the universe was a black hole which was so dense that it could contain hardly anything. Subsets of the universe could contain arbitrary elements. A system built from this universe was based on various relations applied to the subsets. The satisfiability of the system was a crucial issue, but it could be partially ensured by the local consistency of the defined relations, and this could be done independently of the nature of the subset elements. This allowed us to reason about the subsets of the universe at a reasonable cost. The powerset lattice is the mathematical term for the structure of this system.

This thesis proposes a new means to tackle set based combinatorial search problems in a constraint logic programming framework. The main contribution of the work is a new constraint logic programming language allowing set based constraint satisfaction problems to be modelled and solved in an elegant way. We introduce the notion of set domain following the concept of finite integer domain [Fik70]. The elements of a set domain are known sets containing arbitrary values, and the set domain itself represents a powerset. It is defined as a set interval specified by its lower and upper bounds. The constraints of the language are built-in relations applied to variables ranging over set domains. The solver is based on an extension of constraint satisfaction techniques — originating in artificial intelligence— to deal with set constraints.

Domains in CLP

Logic programming [Kow74][CKC83] [Llo87] is a powerful programming framework which enables the user to state nondeterministic programs in relational form. In the recent years, the concept of finite domain [HD86] *i.e.*, set of natural numbers, has been embedded in logic programming to allow for efficient tackling of combinatorial search problems modelled as Constraint Satisfaction Problems (CSP)[Mac77]. A CSP is commonly described by a set of variables ranging over a set of possible values (the domains) and a set of constraints applied to the variables. It is well know that combinatorial search problems are $\mathcal{NP}$-complete [PS82]. The solving of a CSP is based on constraint satisfaction techniques [Mac77][MF85]. They are preprocessing techniques aiming at pruning the search space, associated to a CSP, before the search procedure (eg. backtracking) starts. There are two different uses of these techniques in logic programming coming from two distinct motivations. One consists in programming CSPs and constraint satisfaction techniques at a meta level, with respect to a logic programming language [RM90][MR93]. This approach shows how to use logic programming for solving CSPs and how to transform logic programs using constraint satisfaction techniques. The second aims at extending a logic-based language with constraint satisfaction techniques at the language level [HD86]. This has led to the first development of a Constraint Logic Programming (CLP) language on finite domains, CHIP [DSea88] (Constraint Handling In Prolog).

CHIP extends the application domain of logic programming to the efficient solving of combinatorial search problems. Typical examples are scheduling applications, warehouse location problems, disjunctive scheduling, cutting stock, etc [DSH88a] which come from artificial intelligence or operations research. The success of CHIP prompted the development of new finite domain CLP languages, classified as CLP(FD) languages, but also raised the question of its limitations. Some of the limitations are concerned with the difficulties CLP(FD) languages have to model and solve a class of combinatorial problems based on the search for sets or mapping objects. Set partitioning, set covering, matching problems are such combinatorial search problems. The main motivation of our work is to provide an elegant solution to this problem. So far, a finite domain CSP approach models a set either as a list of variables taking their value from a finite set of integers ($[x_1, ..., x_n], x_i \in \{1, 2, 3, 4\}$), or as a list of 0-1 variables ($[y_1, ..., y_m], y_i \in \{0, 1\}$). The first approach requires the removal of order and multiplicities among the elements of the list, which is achieved by adding ordering constraints ($x_1 < x_2 < ... < x_n$). Constraints over sets are modelled using arithmetic constraints. This is not natural, costly in variables, and this often makes the program non-generic. The second approach, based on the use of 0-1 variables, originates from 0-1 Integer Linear Programming (ILP) [Sch86]. It makes use of the one-to-one correspondence which exists between a subset s of a known set S and a boolean algebra. This correspondence is defined by the characteristic function:

$$f : y_i \longrightarrow \{0, 1\}, f(y_i) = 1 \text{ iff } i \in s, 0 \text{ otherwise}$$

In other words, to each element in S a 0-1 variable is associated, which takes the value 1 if and only if the element belongs to the set s. This approach requires a lot of variables. In addition it does not ease the statement of set constraints such as the set inclusion, because the inclusion of one list into another requires considering a large amount of linear constraints over the 0-1 variables. This is not very natural, nor concise. To cope with this problem, two solutions have been proposed. One consists in defining a class of built-in predicates, referred to as global constraints [Bel90a][BC94], which allow for concise statement and global solving of a collection of constraints. One way to achieve such a global reasoning is to use operations research techniques in a CLP setting. This approach aims both at providing a better pruning of the variable domains by taking into account several constraints at a time. It also extends the programming facilities of CLP(FD) languages to handle efficiently specific problems such as the disjunctive scheduling, the computation of circuits in a graph, etc. *The second solution, presented in this thesis, aims at extending the expressiveness of the language to embedding sets as objects searched for, and to provide set and mapping constraints for general purposes.* This requires investigating how CLP languages based on sets tackle the set satisfiability problem and how well expressiveness can be combined with efficiency.

Sets in LP and CLP

Most of the recent proposals to embed sets as a high level programming abstraction assume a logic-based language as the underlying framework. This is a result of the declarative nature of logic programming which combines well with set constructs, and from its nondeterminism which is suitable for stating set-based programs. For instance, a pure logic programming language is adopted in [BNST91] [Kup90] [DOPR91] [STZ92], an equational logic language in [JP89], and a CLP language in [Wal89] [LL91] [DR93] [BDPR94]. Constraint Logic Programming (CLP) languages dealing with sets, CLP(Sets), are defined as instances of the CLP scheme [JL87] over a specific computation domain describing the class of allowed sets and set constructs. CLP combines the positive features of logic programming with constraint solving techniques. The concept of constraint solving replaces the unification procedure in logic programming and provides, among others, a uniform framework for handling set constraints (eg. $x \in s, s \subseteq s_1, s = s_2$).

The various CLP(Sets) languages aim at modelling and prototyping set based problems in a natural and concise manner. They deal with extensional sets defined by a set constructor (eg. $\{x\} \cup S$, $\{x_1, ..., x_n\}$) such that the set equality is either Associative, Commutative and Idempotent (ACI)[LL91] or commutative and right-associative [DR93]. These properties are defined by axiomatizing a set theory. Regarding the set satisfiability problem, it is $\mathcal{NP}$-complete or even $\mathcal{NP}$-hard [LS76] [PPMK86] [KN86] [Hib95], depending on the class of axioms and predicates considered. In addition, the satisfaction of the ACI axioms introduces non determinism in the unification procedure itself. Each of these languages provides a sound and complete solver. This infers that the complexity of solving set problems in exponential and that the resolution procedure is equivalent to applying an exhaustive search procedure when solving an $\mathcal{NP}$-complete problem. In [LLLH93], variables in a set $\{x_1, ..., x_n\}$ can range over finite domains, and the solver makes use of consistency techniques to prune the domains. This pruning is rather weak since the satisfaction of the ACI axioms introduces another source of non determinism. This prevents us from pruning the search space before the search procedure starts. This lack of efficiency is not a limitation when we take into account the objective of these languages, namely dealing with theorem proving [DR93] or combinatorial problem prototyping [LL91], but it is a problem when from prototyping we move to problem solving.

To achieve a better efficiency, the nondeterministic set unification procedure of constructed sets should be replaced by a deterministic procedure over sets represented as variables. In addition, sets should range over domains so as to make use of preprocessing techniques such as constraint satisfaction techniques. To achieve this, *we propose a language which enables us to model a set-based problem as a set domain CSP —where set variables range over set domains—, and which tackles set constraints by using constraint satisfaction techniques.* A set domain can be a collection of known sets like $\{\{a, b\}, \{c, d\}, \{e\}\}$. It might happen that the elements of the domain are not ordered at all, and thus if large domains are considered, it is not possible to approximate the domain reasoning by an interval reasoning as in some CLP(FD) systems. To cope with this, *we propose to approximate a set domain by a set interval specified by its upper and lower bounds, thus guaranteeing that a partial ordering exists. This allows us to make use of constraint satisfaction techniques by reasoning in terms of interval variations, when dealing with a system of set constraints.* The set interval $[\{\}, \{a, b, c, d, e\}]$ represents the convex closure of the set domain above.

The strengths of handling intervals in CLP have recently been proved when dealing, in particular, with integers and reals. On the one hand, interval reasoning does not guarantee that all the values from a domain are consistent, versus domain reasoning. On the other hand, it removes at a minimal cost some values that can never be part of any feasible solution. This is achieved by pruning the domain bounds instead of considering each domain element one by one. Interval reasoning is particularly suitable to handle monotonic binary constraints (e.g. $x \leq y, s \subseteq s_1$), where it guarantees the correctness properties of domain reasoning while being more efficient in terms of time complexity.

Intervals in CLP

The introduction of real intervals into CLP aims at avoiding the errors resulting from finite precision of reals in computers. A real interval is an approximation of a real and is specified by its lower and upper bounds. It does not denote the set of possible values a variable could take but a variation of an infinite number of values. Cleary [Cle87] introduced a relational arithmetic of real intervals in logic programming based on the interpretation of arithmetic expressions as relations. Such relations are handled by making use of projection functions and closure operations, which correspond to the definition of transformation rules expressing each real interval in terms of the other intervals involved in the relation. These transformation rules approximate the usual consistency notions [Mac77]. The handling of these rules is done by a relaxation algorithm which resembles the arc-consistency algorithm AC-3 [Mac77]. This approach prompted the development of the class of CLP(Intervals). A formalization of this approach is given in [Ben95].

While CLP(Intervals) languages make use of constraint satisfaction techniques, they do not model CSPs because the solving of a problem modelled in a CLP(Intervals) language searches for the smallest real intervals such that the computations are correct. It guarantees that the values which have been removed are irrelevant, but does not bound the real variables to a value. On the one hand, set intervals in constraint logic programming resemble the real interval arithmetic approach in terms of interpreting set expressions as relations and using interval reasoning to perform set interval calculus when handling the constraints. We make use of similar projection functions which are the only way to handle set expressions (e.g. $s \cup s_1, s \cap s_1$) as relations. We also approximate the set domain of a set expression by a convex interval. On the other hand, *set intervals in constraint logic programming contribute to the definition of a language which allows one to model and solve discrete CSPs in the CLP framework.* In practice, this corresponds to providing a labelling procedure in order to reach a complete solution. Regarding optimization problems, it is necessary to allow the definition of cost functions which necessarily deal with quantifiable, *i.e.*, arithmetic, terms. This requires the definition of a class of functions, interpreted as constraints, which map sets to integers (e.g. the set cardinality), and a cooperation between two solvers (set solver and finite domain solver). These requirements differ from that of CLP(Intervals) languages where the completeness issue is still an open problem because of the infinite size of real intervals.

Short outline

Part I- Sets and Intervals in CLP languages.
This part has three sections. *Section 1* presents the constraint logic programming scheme and its operational model. *Section 2* presents the class of CLP(Sets) languages, with a particular attention given to the relationship between their application domains and their constraint solvers. *Section 3* surveys consistency notions and algorithms and describes their embedding into the class of CLP(FD)

and CLP(Intervals) languages.

Part II- The Language.
This is the central part of the dissertation. This part contains three main sections. *Section 4* describes the formal framework of a constraint logic programming language over set domains. It comprises the description of the system, that is the constraint domain —over which set interval calculus is performed— and the operational semantics. *Section 5* describes the CLP language over set domains, called Conjunto, which we have designed and implemented using the constraint logic programming platform ECL^{i}PS^e [ECR94]. This section shows how constraint satisfaction techniques can be adapted to deal with constraints over set intervals using interval narrowing techniques. *Section 6* presents applications developed in Conjunto. The applications illustrate the modelling facilities of the language and its ability to solve in an efficient way large problems. Comparative studies are made with finite domain CSP approaches.

Conclusion.
In this part, we give an evaluation of the results achieved, present the related lines of work, and discuss further possible research in terms of improving the current kernel and designing further extensions.

Part I

Sets and Intervals in CLP Languages

1
Constraint Logic Programming

Constraint logic programming is a relatively new programming framework (1987) which aims at extending the applicability of logic programming to mathematical calculus (arithmetic calculus, set calculus, etc). Constraint logic programming defines a class of languages CLP(X) parameterized by their computation domain X (eg. finite domains, reals, sets). This chapter describes the constraint logic programming scheme, the constraint solving paradigm, and gives a short overview of the computation domains which currently exist.

1.1 A logic-based language

A Constraint Logic Programming (CLP) language is a logic-based language [Kow74] [CKPR73] (cf. Prolog [CKC83]) that is a nondeterministic programming language where procedures are defined in a relational form. The syntax of a CLP program is that of a logic-based program based on a collection of Horn clauses [Llo87].

Definition 1 *A Horn clause is a disjunction of atoms with at most one non-negated atom:*

$$(Q_1 \vee \neg P_1 \vee ... \vee \neg P_n) \textit{ or } (Q_1 \leftarrow P_1 \wedge ... \wedge P_n)$$

where Q_1 and the P_i are atoms and the variables appearing in the atoms are assumed to be universally quantified.

The declarative interpretation of a Horn clause corresponds to:

Q_1 is true if $P_1, ...P_n$ are true

A program goal is a clause of the form: $\leftarrow G_1, G_2, ...G_n$ where the G_i are atoms.

To distinguish atoms from constraint relations, a CLP program is formally defined by a collection of rules of the form

$$Q_1 \leftarrow C_1 \wedge ... \wedge C_n \diamond P_1 \wedge ... \wedge P_m$$

where Q_1 is an atom, the P_i are atoms and the C_i are constraints. A goal is a collection of constraints and atoms, and corresponds to a rule without head, here without Q_1.

1.2 The CLP scheme

The CLP scheme defined by Jaffar and Lassez [JL87] describes a formal semantics which subsumes logic programming. CLP defines a class of languages parameterized by their computation domain. A CLP language is characterized by its computation domain, its set of allowed constraints, and its constraint solver. The CLP scheme defines the following properties to be satisfied by a constraint logic programming language CLP(X) which is an instance of the scheme.

Let us consider the computation domain $\mathcal{D}$ and the set $\mathcal{L}$ of constraints. The structure ($\mathcal{D}$, $\mathcal{L}$) describes the constraint domain over which constraint solving is performed. This structure must have a compactness property which guarantees that every element from the underlying computation domain is finitely definable using the constraints of the constraint domain. Consider a theory $\mathcal{T}$ which axiomatizes some of the properties of constraints in $\mathcal{L}$ applied to elements from $\mathcal{D}$. The formal semantics defined in the CLP scheme describes the algebraic semantics of the language and its correspondence with the logical semantics. Jaffar and Lassez introduced new concepts to deal with the constraint domain structure and the theory which are the ones of *solution-compact* structure and a *satisfaction complete* theory.

Definition 2 *A structure ($\mathcal{D}$, $\mathcal{L}$) is* solution-compact *if every element in $\mathcal{D}$ is the unique solution of a finite or infinite set of contraints in $\mathcal{L}$, and every element in the complement of the solution space of a constraint c belongs to the disjoint solution space of some finite or infinite family c_i of constraints.*

The correspondence between the theory $\mathcal{T}$ and the computation domain $\mathcal{D}$ aims at ensuring that $\mathcal{T}$ and $\mathcal{D}$ correspond on the satisfiability of elements from $\mathcal{L}$ and that every unsatisfiable constraints in $\mathcal{D}$ is also detected by $\mathcal{T}$. This is defined by the three following conditions:

- $\mathcal{D}$ is a model of $\mathcal{T}$.
- for every constraint $c \in \mathcal{L}, \mathcal{D} \models \exists c$ iff $\mathcal{T} \models \exists c$
- $\mathcal{T}$ is *satisfaction-complete* with respect to $\mathcal{L}$ if for every constraint $c \in \mathcal{L}$, c is either provably true or false $\mathcal{T} \models \exists c$ or $\mathcal{T} \models \neg\exists c$.

If we consider a CLP program P and a goal G, the logic programming inference mechanism searches for a substitution σ such that $G\sigma$ (possibly infinite set of instances) is a logical consequence of P. A CLP goal can also be seen as a logical consequence of a program, provided it is also a logical consequence of the theory $\mathcal{T}$. Considering the C_i as a conjunction of constraints, we have:

$$P, \mathcal{T} \models \forall(C_1, C_2, ..., C_n \Rightarrow G)$$

The satisfaction-complete property of the theory plays a role in the completeness of the constraint solver. In practice it turns out that efficient constraint solving methods over certain structures cannot be combined with completeness of the solver. Indeed, as soon as the satisfiability problem over a computation domain (eg. finite domains, sets) is an $\mathcal{NP}$-complete problem, the special purpose constraint solver will necessarily take exponential time to guarantee completeness of the satisfaction procedure. For efficiency reasons, some solvers achieve a partial constraint solving based on consistency techniques. These solvers will be subsequently described.

Recently, Saraswat et al. [SRP91] proposed a generalization of the CLP scheme which defines the framework of concurrent constraint programming. It is based on two operators *ask* & *tell* which correspond respectively to constraint entailment and contraint statement actions. This framework has been adapted by Van Hentenryck and Deville to formalize the incompleteness of some constraint solvers dealing with linear constraints over finite domains [HD91] [HSD93].

1.3 Constraint solving

CLP is a generalization of LP where unification —the basic operation of LP languages— is replaced by constraint solving techniques. With regard to the constraint solving mechanism of a CLP program, Colmerauer [Col87] defined a general operational semantics which establishes an analogy with the SLD-resolution procedure embedded in LP. The SLD-resolution takes as input a set of clauses. It unifies the expressions, stores a sequence of substitutions and returns as output the successful substitution. The resolution of CLP goals replaces unification by constraint solving and returns as output a set of satisfiable constraints. It can be defined as a transition system on states comprising goals and constraints. Each transition rule can be interpreted as a rewriting process which derives a new state from the previous one. A solution is found when the final state does not contain goals to be solved. In case the set of constraints is not satisfiable, the resolution fails.

One computation step of the constraint resolution procedure can be represented by analogy with one SLD-derivation step as follows. C, C_1 are sets of constraints, G, B sets of predicates (or terms) and a one predicate. It is depicted in the figure 1.1.

from $\leftarrow C \diamond G$ and $a \leftarrow C_1 \diamond B$
infer $\leftarrow C_2 \diamond (\downarrow G \cup B)$
if merge $(\{\uparrow G = a\} \cup C_1 \cup C)$ into C_2

$\uparrow G$ represents the first left atom in G, $\downarrow G$ represents the remainder (cf. [Coh90])

Figure 1.1 Constraint Solving: one resolution step

The merge function is the essential one in the solver. It checks that the new constraints related to the goal are satisfiable in conjunction with the current ones. If this new set of constraints is not satisfiable the procedure fails, otherwise it returns the simplified set of constraints. This approach was originally defined for the CLP language Prolog III [Col87] [Col90].

However this function, which embeds two actions (satisfiability checking and simplification process), can not be applied for constraint solvers that only ensure partial constraint solving and make use of delay mechanisms (where a constraint is neither considered satisfiable nor simplified). Consequently, it can not be generalized to any special-purpose constraint solver. Several operational models have been defined for specific constraint domains. Jaffar and Maher [JM94] proposed a fairly general framework also based on the transition system on states, but which splits the merge function into several functions each of which derives distinct transitions corresponding to a resolution (simplification), an addition of new constraints, a consistency checking, etc. It also distinguishes between active and passive constraints. The active constraints are those which can lead to simplifications, and the passive ones are those which can only be checked, but might become active once they are completely solved.

1.4 Constraint domains

Various computation and more exactly constraint domains have been investigated in recent years, but only some of them will been mentioned here. A more detailed description can be found in [JM94]. The most widely known are:

- Linear rational arithmetic (CHIP [DSea88], Prolog III [Col87], Prolog IV [BT95]) and real arithmetic (CLP($\mathcal{R}$) [JM87]). Their solvers are based on the simplex algorithm, generalized to take into account handling of disequations and incrementality of the solving.
- Boolean algebra (CHIP, Prolog III), whose solvers are based, respectively, on Boolean unification and on a combination of the SL resolution and saturation.
- Linear arithmetic over finite domains (CHIP and others), whose solver is based on consistency techniques.
- Real intervals (eg. BNR-Prolog [OV90], CLP(BNR) [OB93], Interlog [Lho93], Prolog IV, ICHIP [LvE93] Newton [BMH94]) whose solvers adapt consistency techniques to perform interval reasoning.
- Set calculus ({log} [DOPR91] [DR93], CLPS [LL91]) and regular sets (CLP(Σ^*) [Wal89]). These languages aim at guaranteeing the soundness and completeness of their respective solvers. They deal with set constructs, and provide a collection of allowed set operations and constraints.

The last three constraint domains, which have some common points with our work are presented in the next two sections.

2 Sets in CLP

Most of the recent proposals to embed sets as a high level programming abstraction assume a logic-based language as the underlying framework. It follows from to the declarative nature of logic programming, which well combine with set constructs, and its nondeterminism which is suited to stating set-based programs. This chapter describes the class of CLP(Sets) languages which embed sets in constraint logic programming. Particular attention is put into the description of the computation domains and the constraint solvers of CLP(Σ^*) which deals with regular sets, {log} (reviewed from a LP to a CLP point of view) which axiomatizes a set theory and CLPS which aims at prototyping combinatorial problems using sets, multisets and sequences.

2.1 CLP(Σ^*)

CLP(Σ^*) [Wal89] represents an instance of the CLP scheme over the computation domain of regular sets. A regular set is a finite set composed of strings which are generated from a finite alphabet Σ. CLP(Σ^*) has been designed and implemented to provide a logic-based formalism for incorporating strings into logic programming in a more expressive manner than the standard string-handling features (eg. concat, substring). A CLP(Σ^*) program is a Prolog program enriched with regular set terms and built-in constraints.

Operations on regular sets comprise concatenation $R_1.R_2$, disjunction or union $R_1 + R_2$ (*i.e.*, $R_1 \cup R_2$) and the closure operator R_1^* which describes the least set R' such that $R' = \epsilon + (R', R_1)$. These operations allow us to build any regular expression when combined with the identity elements under concatenation (1) and union ($\emptyset$). This language provides an atomic constraint over set expressions which is the membership constraint of the form x `in` e where x is either a variable or a string and e is a regular expression. For example A `in` $(X.''ab''.Y)$ states that any string assigned to variable A must contain the substring ab.

Overview of the solver The constraint paradigm allows to replace the unification procedure by constraint solving in the computation domain. The satisfiability of membership constraints over regular sets clearly poses the problem of termination. In the above example, if Y is a free variable there is an infinite number of instances for A. The solver guarantees termination by: (i) applying a scheduling strategy which selects the constraints capable of generating a finite number of instances, (ii) applying a satisfiability procedure based on deduction rules which check and transform the selected atomic constraints. The non selected ones are simply floundered.

The selected constraints x `in` e are such that either e is a string or e is a variable and x a string. The conditional deduction rules over each of these constraints infer a new constraint or a simplified one if a given condition is satisfied. Each condition represents a possible form of selected set constraints.

$$\frac{\begin{pmatrix} w = w_1.w_2 \\ \sigma_1 \vdash \ "w_1" \ \mathtt{in} \ e_1 \\ \sigma_2 \vdash \ "w_2" \ \mathtt{in} \ e_2 \end{pmatrix}}{\sigma_1 \cup \sigma_2 \vdash \ "w" \ \mathtt{in} \ e_1.e_2} \quad \text{and} \quad \frac{\begin{pmatrix} \sigma_1 \vdash X_1 \ \mathtt{in} \ e_1 \\ \sigma_2 \vdash X_2 \ \mathtt{in} \ e_2 \end{pmatrix}}{[X = (X_1\sigma_1).(X_2\sigma_2)] \vdash X \ \mathtt{in} \ e_1.e_2}$$

The σ_i are idempotent substitutions, which means that given two substitutions σ_1 and σ_2, $\sigma_1 \cup \sigma_2$ produces the most general idempotent substitution if one exists that is more specific than the two previous ones.

Soundness and completeness of the deduction rules are guaranteed only if there are no variables within the scope of any closure expression e^* in addition to the criteria of constraint selection.

This approach constitutes a first attempt to compute regular sets by means of constraints like the membership relation. The complexity of the satisfiability procedure is not given, but infinite computations are avoided thanks to the use of floundering.

2.2 CLPS

The CLPS [LL91] [LL92] language is a CLP language based on a three sorted logic. The three sorts correspond to sets, multi-sets and sequences of finite depth (eg. $s = \{\{\{e, a\}\}, c\}$ is a set of depth three). The concept of depth is equivalent for each sort. Atomic elements can be any Herbrand term, arithmetic expression or integer domain variable. Set expressions are built from the usual set operator symbols $(\cup, \cap, \setminus, \#)$. Set variables are constructed either iteratively by means of the set constructor $\{x\} \cup s$ or by extension by grouping elements within braces (eg. $\{x_1, ..., x_n\}$). The language also embeds finite integer domains and allows set elements to range over a finite domain. Sequences and multi-sets are built using, respectively, the constructors $sq\{...\}$ and $m\{...\}$. Basic constraints (implemented in the language) are relations from $\{\in, =, \notin, \neq, \subseteq\}$ interpreted in the usual mathematical way together with a depth (::) and a type checking operator. Note that set equality relation sould be associative, commutative and idempotent. These properties are specified by the ACI notation [LS78].

The satisfiability problem for sets, sequences and multisets is $\mathcal{NP}$-complete [LS76]. To cope with this, CLPS provides several methods whose use depends on the characteristics of the CLPS program at hand.

Overview of the solver The CLPS solver makes use of various techniques comprising: (i) a set of semantical-consistency rules, (ii) an arc-consistency algorithm of type AC-3 [Mac77] combined with a local search procedure (forward checking) and (iii) a transformation procedure. The rules in (i) check the consistency of each set constraint with respect to homogeneity of types, depth and cardinality. For example the system

$$\{x\} = \{y, z\}$$

is semantically-consistent if $y = z$.

A semantically-consistent system of set constraints is then solved in two stages. The solver first divides the system in two independent subsets. One, written SC_{fd} contains set constraints whose constrained sets are sets of integer domains variables. The other one, written SC_v contains sets and set constraints where set elements are free variables or known values. The solver applies (ii) and (iii) respectively to check satisfiability over SC_{fd} and SC_v:

- A system SC_{fd} is consistent if each of the set constraints it contains is arc consistent. This is achieved by removing all values from the domains of the set elements which cannot be part of any feasible solution. For example, the above system is consistent if $x \in \{2, 3\}$ and $[y, z] \in \{2, 3\}$. Completeness of the resolution is guaranteed by the labelling procedure which performs forward checking combined with the first fail principle. It amounts to assigning a value to a set element from its domain and to inferring possible new domain modifications. However, it might happen that due to the ACI properties of set equality, distinct selected values for the elements will generate identical values for the sets. This nondeterminism in the unification of constructed sets requires in the worst case an exponential number of choices to be made. The system $[x_1, ..., x_n] \in \{1, ..., m\}$, $\{x_1, ...x_n\} = \{1, ..., m\}$ corresponds to 2^{n-m} computable solutions.

- A system SC_v is satisfiable if its equivalent integer linear programming form is satisfiable. To check satisfiability, the system provides a correct and complete procedure which transforms the set constraint system into an equivalent mathematical model based on integer linear programming [HLL93]. This procedure consists in flattening each set constraint and reducing the system of flattened formulas to an equivalent system of linear equations and disequations over finite domain variables. The derived system is then solved using consistency techniques. The flattening algorithm works by adding additional variables to reach forms from ($x = y,\ x \in y,\ x = \{x_1, ..., x_n\}, x = y \cup z,\ x = y \cap z,\ x = y \setminus z$, etc.). The reduction to linear form is performed by associating to each set variable x_i a new variable C_{xi} which represents its cardinality and to each pair of variables $(x_i,\ x_j)$ a new binary variable Q_{ij} denoting possible set equality constraints. If there are n constraints the complexity of the reduction procedure is in $\mathcal{O}(n^3)$ [HLL94] [Hib95].

The proposed solving methods are among the most appropriate for handling set constraints over constructed sets. They fit the application domain of the language which aims at prototyping combinatorial search problem dealing with sets, multi-sets, or sequences. Unfortunately the nondeterminism in the unification of set constructs prevents an efficient pruning of the domains attached to set elements (in case they represent domain variables). The focus is put on the expressive power of the language rather than on the efficient solving.

2.3 {log}

{log} [DR93] is an instance of the CLP scheme designed and implemented mainly for theorem proving. It embeds an axiomatized set theory whose properties guarantee soundness and completeness of the language. Set terms are constructed using the interpreted functors `with` and {}, e.g. $\emptyset$ `with` x `with` ($\emptyset$ `with` y `with` z) = {{z,y},x}. The language includes a limited collection of predicates ($\in, =, \neq, \notin$) as set constraints. The axiomatized set theory consists of a set of axioms which describe the behaviour of the constructor *with*. For example the extensionality axiom shows how to decide if two sets can be considered equal:

$$
\begin{aligned}
v \ \texttt{with}\ x = w\ \texttt{with}\ y \rightarrow\ & \\
& (x = y \wedge v = w) \vee (x = y \wedge v\ \texttt{with}\ x = w) \vee \\
& (x = y \wedge v = w\ \texttt{with}\ y) \vee \exists z\ (v = z\ \texttt{with}\ y \wedge w = z\ \texttt{with}\ x)
\end{aligned}
$$

Using the axioms, a set of properties are derived describing the permutativity (right associativity) and absorption of the `with` constructor.

For example, the permutativity property is depicted by:

$(x$ `with` $y)$ `with` $z = (x$ `with` $z)$ `with` y (permutativity)

Overview of the solver The complete solver consists of a constraint simplification algorithm defined by a set of derivation rules with respect to each primitive constraint. A derivation rule for the equality constraint is, for example:

h `with` $\{t_n, ..., t_0\} = k$ `with` $\{s_m, ..., s_o\}$

If h and k are not the same variables then select non-deterministically one action among:

- $t_0 = s_0$ and h `with` $\{t_n, ..., t_1\} = k$ `with` $\{s_m, ..., s_1\}$
- $t_0 = s_0$ and h `with` $\{t_n, ..., t_0\} = k$ `with` $\{s_m, ..., s_1\}$
- $t_0 = s_0$ and h `with` $\{t_n, ..., t_1\} = k$ `with` $\{s_m, ..., s_0\}$
- h `with` $\{t_n, ..., t_1\} = N$ `with` s_0, N `with` $s_0 = k$ `with` $\{s_m, ..., s_1\}$

otherwise select i in $\{0, ..., m\}$ and apply one action from another set of rules.

This non deterministic satisfaction procedure reduces a given constraint to a collection of constraints in a suitable form by introducing choice points in the constraint graph itself. This leads to a hidden exponential growth in the search tree, since in the worst case all computable solutions have to be investigated (if $s_1 = s_2$ and $\#s_1 = n$, there are 2^n computable solutions). But completeness is required if one aims at performing theorem proving. Thus, there is no possible compromise here between completeness and efficiency.

A recent extension to the language introduces intensional sets in constraint logic programming [BDPR94]. Allowing for set grouping capabilities, the intensional definition is handled by reducing the set grouping problem to the problem of dealing with normal logic programs, *i.e.*, programs containing negation in the body of the clauses.

3 Interval and domain reasoning in CLP

Two classes of CLP languages deal with variables ranging over intervals and/or finite domains. The class of CLP(FD) languages dealing with finite integer domains considers linear arithmetic over natural numbers as well as some symbolic constraints, provided that the variables take their value from a finite set of integers. They aim at modelling and solving constraint satisfaction problems in a constraint logic programming framework. The second class is that of CLP(Intervals) languages which deal with real interval arithmetic. The use of intervals is meant to approximate real numbers so as to avoid rounding errors. This chapter describes these two classes of languages, whose solvers are based on consistency techniques.

3.1 Constraint satisfaction problems

Formally, a Constraint Satisfaction Problem (CSP) is a tuple <V,D,C> where:

- V is a set of variables $\{V_1, ..., V_n\}$,
- D is a set of domains $\{D_1, ..., D_n\}$ where D_i is the domain associated to the variable V_i,
- C is a set of constraints $\{C_1, ..., C_m\}$ where a constraint C_j involves a subset of the variables.

The constraint set in a CSP is such that each variable appearing in a constraint should take its value from a given domain. The constraint set is often represented by a constraint network whose nodes are the variables with their associated domains and whose arcs are the constraints. A CSP models $\mathcal{NP}$-complete problems as search problems where the corresponding search space is represented by a Cartesian product space $D_1 \times D_2 \times ... \times D_n$ of the domains (cf. Golomb [GB65]).

3.2 Constraint satisfaction

The solution of a CSP is a set (or subset as noted in [MR93]) of variables assigned to one value. The solving of a CSP amounts first to applying a set of preprocessing methods referred to as consistency techniques and then applying some search techniques or labelling procedure. Consistency techniques aim at pruning the search space before a standard search procedure like backtracking is applied and thus at improving the average complexity of standard backtracking [Wal60][GB65]. Consider a search tree as the abstract representation of an $\mathcal{NP}$-complete problem where one branch is a combination of values. Backtrack programming [Flo67] aims at computing a feasible solution (or all solutions) of such problems using an exhaustive searching process. This process explores all the branches and stops searching one branch as soon as it encounters a failure.

Formally, the backtracking algorithm aims at finding a solution specified by a vector $(x_1, x_2, ..., x_n)$ with $x_i \in D_i$ such that it satisfies a set of constraints represented by a "criterion function" $\phi(x_1, x_2, ..., x_n)$. The solution vector might not be unique, and it may suffice to find one such vector or be necessary to find all of them depending on the problem. The criterion function is usually two valued (true or false). If a partial vector $(x_1, x_2, -, -, ..., -)$ is unacceptable, all possible solutions containing x_1 and x_2 can be ruled out without having to be considered individually. The search is stopped in this branch.

Non deterministic algorithms are convenient representations of systematic search procedures, but they turn out to be inefficient for large problems. Exhaustive search combined with the thrashing[1] phenomenon leads in the general case to computations that are exponential in the size of the Cartesian product of the domains. A solution to this problem consists in removing inconsistent values before any attempt is made to include them in the sample vector. This preprocessing step is achieved by consistency techniques.

The current consistency algorithms perform different degrees of preprocessing. Their behaviour amounts to going through the constraint network in a node-driven way and checking among other methods, the consistency of each node (node consistency), of each arc (arc-consistency) [Ull66] [Fik70] [Wal75] [Mac77], of each path of length two [Mon74] [Mac77] [MH86], of each path of length k [Fre78].

The definitions of node, arc and path consistency are usually given for unary and binary constraints denoted, respectively, $P_k(x_k)$, $P_{ij}(x_i, x_j)$. This restriction does not prevent consistency techniques from being applied to n-ary constraints,

[1] Thrashing means that some unacceptable values will be considered at several steps of the search even though they can never be part of any feasible solution.

since any n-ary constraint can be expressed in terms of binary ones. Let us recall the definitions of node, arc and path consistency [Mac77].

Definition 3 *A node i is* node consistent *if and only if for any value $x \in D_i$, $P_i(x)$ is true.*

Definition 4 *An arc (i, j) is* arc consistent *if and only if for any value $x \in D_i$ such that $P_i(x)$, there is a value $y \in D_j$ such that $P_j(y)$ and $P_{ij}(x, y)$.*

Definition 5 *A path of length m through the nodes $(i_0, i_1, ..., i_m)$ is* path consistent *if and only if for any values $x \in D_{i_0}$ and $y \in D_{i_m}$ such that $P_{i_0}(x)$, $P_{i_m}(y)$ and $P_{i_0,i_m}(x, y)$ hold, there is a sequence of values $z_1 \in D_{i_1}, ...z_{m-1} \in D_{i_{m-1}}$ such that:*

(i) $P_{i_1}(z_1)$ and ... and $P_{i_{m-1}}(z_{m-1})$ hold,
(ii) $P_{i_0,i_1}(x, z_1)$ and $P_{i_1,i_2}(z_1, z_2)$ and... and $P_{i_{m-1},i_m}(z_{m-1}, y)$

These definitions can be generalized to the notions of node, arc or path consistency of a constraint network, which correspond to having every node, arc or path —in the corresponding directed graph— consistent.

3.2.1 Algorithms

The node consistency algorithm checks that, for each variable V_i appearing in an unary constraint $P_i(V_i)$, all the elements x in its domain D_i satisfy the constraint $P_i(x)$. If some elements do not satisfy this constraint, they are removed from the domain D_i. This algorithm is quite simple and requires a single pass through all the unary constraints (cf. [Mac77]).

The various arc-consistency algorithms require more complex processing. They are based on the following observation (cf. [Fik70]): *if for some $x \in D_i$ there is no $y \in D_j$ such that $P_{ij}(x, y)$ holds then x should be removed from the domain D_i. This test should be done for each $x \in D_i$ to conclude if one arc is consistent or not.* This test resembles the criterion function used in backtrack programming, but it differs in that it does not choose a value x for a variable but it *tests* if this value is an acceptable one. Thus the assignment process is replaced by a test. The result of this action should be an answer to whether the domain D_i has been modified. This action, depicted in the $REVISE((i, j))$ procedure in figure 3.1, is the kernel of current arc consistency algorithms.

procedure REVISE((i,j)):
begin
  DELETE $\leftarrow$ **false**
  for each $x \in D_i$ **do**
    if there is no $y \in D_j$ such that $P_{ij}(x,y)$ **then**
      begin
        delete x from D_i;
        DELETE $\leftarrow$ **true**
      end;
  return DELETE
end

Figure 3.1 REVISE procedure

$REVISE((i,j))$ returns the answer to whether a domain modification was required to infer arc consistency of a binary constraint. When the arc consistency of the constraint network is concerned, the process is more complicated. Indeed, checking an arc (j,k) might require revisions of the domain of j and consequently, the as yet consistent arc (i,j) might not be consistent anymore. Therefore, as opposed to the node consistency algorithm, arc consistency over a network can seldom be achieved in a single pass through all the arcs. At this point the various arc consistency algorithms proposed so far differ. The problem is to reconsider as few arcs as possible for complexity, and thus for efficiency reasons. The various generic arc consistency algorithms developed so far are (the first three have been called AC-1 AC-2 AC-3 by Mackworth in [MF85]):

- **AC-1** (embedded in the first constraint system REF-ARF [Fik70]). This is the simplest algorithm. It repeatedly passes through all the arcs each time one domain is revised until there is no change on an entire pass. At this point the network must be consistent. This approach is intuitive but obviously inefficient, because a single modification of the domain causes all the arcs to be revised, whereas only a subset of them might be affected.

- **AC-2** (based in spirit on Waltz's filtering [Wal75]). Noting the weaknesses of AC-1, Waltz's idea was that arc consistency can be achieved in one pass over all the nodes by taking into account the order of the nodes covered and by ensuring that when a node i is considered in an arc (i,j), all the arcs (g,h) where $g,h \leq i$ must have previously been made consistent. The crucial improvement is that when a node j is considered, all the arcs *leading* from it and to it may have become inconsistent and must be revised again.

- **AC-3** (proposed by Mackworth [Mac77]). This approach moves from the node-driven reasoning of AC-2 to an arc-driven reasoning. All the arcs are stored in one queue and REVISE is applied to each of them sequentially. The basic idea consists in selecting and removing one arc (i, j) from the queue, applying REVISE to it and if the answer is yes (D_i modified) adding to the queue all those arcs $\{(k, i)\}$ that might need to be reconsidered. This algorithm is so far the principal one embedded in most constraint satisfaction solvers. Its description is given below.

- **AC-4** (proposed by Mohr and Henderson [MH86] and based on the techniques developed in the constraint satisfaction system ALICE [Lau78]). While AC-3 is driven by arcs (i, j), AC-4 reasons over arcs (i, c) where i is a node and c an inconsistent value in the domain of i. It moves from handling domains of variables (eg. D_i, D_j) to dealing with inconsistent values associated to one domain. This comes together with the storing of a counter which represents the number of possible values of j such that for each value $b \in D_i$, (b, j) holds. This counter, associated to each string [(i, j),b], is decremented each time an arc (e.g. (b, c)) becomes inconsistent. The basic idea consists in handling the set of arcs $\{(i, c)\}$ for each $c \in D_j$ as well as the number of values which are consistent with one specific value. This approach leads to the optimal arc consistency algorithm with respect to time complexity. However it might be costly in memory utilization due to the amount of information it has to maintain.

- **AC-5** (proposed by Van Hentenryck and Deville [HDT92]). In contrast to the previous algorithm, this one aims at reducing the time complexity by considering the semantics of the constraints at hand. It distinguishes predicates according to their underlying properties (functional, anti-functional, monotonic, etc...). While AC-4 deals with arcs $(i.c)$, AC-5 manipulates elements $< (i, j), v >$ where (i, j) is an arc and v is a value removed from D_j which requires reconsideration of (i, j). Optimal procedures for the class of functional, anti-functional and monotonic constraints have been proposed. The specific case of the monotonic constraints permits performing reasoning over the domain bounds only (assuming the domains are totally ordered). Like AC-4, AC-5 propagates unconsistent values, but instead of decrementing a counter attached to each possible value of one node it adds to the list of the elements all those which correspond to newly inconsistent arcs with respect to one value. The interesting point in both AC-4 and AC-5 is that only necessary information is propagated.

Complexity issues. Mackworth and Freuder show the complexity of AC-1, AC-2 and AC-3 in [MF85]. Let us consider a to be the largest domain size, e the number of arcs and n the number of nodes. The results in the figure below are given in terms of worst case time complexity.

NC	AC-1	AC-2	AC-3	AC-4	AC-5
$\mathcal{O}(an)$	$\mathcal{O}(a^3ne)$	cf. AC-3	$\mathcal{O}(a^3e)^*$	$\mathcal{O}(ea^2)$	$\mathcal{O}(ea^2)^{**}$

Figure 3.2 Complexity of AC algorithms

$*$ This complexity result assumes that the constraint network is connected (it implies $e \geq n-1$).

$**$ This time complexity can be reduced to $\mathcal{O}(ea)$ for the class of functional, anti-functional and monotonic constraints, and their generalization to piecewise functional, anti-functional and monotonic constraints (see [HDT92] for the definition of piecewise decomposition of constraints).

The following AC-3 algorithm is the one upon which most improvements and variations of arc consistency algorithms have been performed. The set of arcs in the constraint graph G is marked by arcs(G) in figure 3.3.

begin
    for $i \leftarrow 1$ **until** n **do** node consistency;
    $Q \leftarrow \{(i,j) \mid (i,j) \in \text{ arcs } (G), i \neq j\}$
    while Q not empty **do**
        begin
            select and delete any arc (k,m) from Q;
            if REVISE $((k,m))$ **then**
                $Q \leftarrow Q \cup \{(i,k) \mid (i,k) \in \text{ arcs}(G), i \neq k, i \neq m\}$
        end;
end

Figure 3.3 AC-3 algorithm

Other approaches toward efficient algorithms are based on the study of the topology of the constraint graph itself (see [Fre82] [Nad88] [RM89]). But these methods have not been embedded so far in CLP solvers, possibly because of the particular properties of the constraint graph they require —which are seldom those of a CSP program.

3.2.2 Search Techniques

While consistency techniques aim at filtering the domains before starting the search, the search techniques embed various degrees of arc-consistency within a standard backtracking procedure. They correspond to the notions of forward checking, partial lookahead and full lookahead [McG79] [HE80]. The pruning achieved by these search techniques ranges between that of backtracking and that of arc consistency.

Consider the initial description of the backtracking process, based on a Cartesian product space $D_1 \times D_2 \times ... \times D_n$, a solution or sample vector $(x_1, ..., x_n)$ where $x_i \in D_i$, and a criterion function to be satisfied $\phi(x_1, ..., x_n)$. A step k in the computation is denoted $(x_1, ..., x_k, -, ..., -)$ which corresponds to having the partial state $(x_1, ..., x_k)$ locally consistent.

Forward checking. Whenever a value x_{k+1} is successfully added to the current state of the sample vector $(x_1, ..., x_k, -, ..., -)$ (*i.e.* we have $\phi(x_1, ..., x_k, x_{k+1}, -, ..., -) = 1$), the domains $D_{k+1}, ..., D_n$ of all as yet uninstantiated variables are filtered to contain only those values that are relevant with this new instantiation. This can be represented by the rule:

$$\forall l \in \{k+2, ..., n\} \forall x_l \in X_l \text{ such that } \phi(x_1, ..., x_k, x_{k+1}, -, ...-, , x_l, -, ..., -) = 1$$

If the domain of any of these uninstantiated variables becomes empty, the constraint fails and backtracking occurs. This method adds to standard backtracking a preprocessing step in which some irrelevant values are removed before they may be taken into account. These values will come only from the domain of each variable directly connected with x_{k+1}.

Full lookahead. Whenever a value x_{k+1} is successfully added to the current state of the sample vector $(x_1, ..., x_k, -, ..., -)$ the forward checking conditions must be satisfied and the domain of each variable —as yet uninstantiated— must be filtered, so that it should only contain those values which are relevant with respect to at least one value in *all* the domains of the variables they are connected with. This is described by the rule:

$\forall\ l \in \{k+1, ..., n\},\ \forall\ x_l \in X_l$ such that:
$\forall\ m \in \{k+1, ..., n\},\ m \neq l,\ \ \exists x_m \in X_m$ which satisfies
$\phi(x_1, ..., x_k, x_{k+1}, -, ..., -, x_l, -, ..., -, x_m, -, ..., -) = 1$

The full lookahead method performs less pruning than arc consistency algorithms because it performs one single pass through all the binary constraints. A consistent arc will never be reconsidered whatever new refinements of the domains of the variables involved may have been performed.

Partial lookahead. This method has been introduced by Haralick [HE80] to augment the filtering process achieved by the forward checking method. It acts some where in between forward checking and full lookahead. The basic idea is not to filter one X_l by considering all the other variables as yet uninstantiated, but to consider only those that are ahead of X_l, which falls as:

$\forall\ l \in \{k+1, ..., n-1\},\ \forall\ x_l \in X_l$ such that:
$\forall m \in \{l+1, ..., n\},\ \ \exists x_m \in X_m$ which satisfies
$\phi(x_1, ..., x_k, x_{k+1}, -, ..., -, x_l, -, ..., -, x_m, -, ..., -) = 1$

3.3 Constraint satisfaction in LP

The solving of CSPs using Logic Programming (LP) has been investigated from two different perspectives. One, proposed by Montanari and Rossi, aims at defining a CSP as a logic program and defining the constraint satisfaction or relaxation algorithm [RM90] [MR91] at a meta level. This approach shows that modelling CSPs and consistency algorithms in LP is adequate. It also shows how logic programs can be transformed and simplified using relaxation algorithms. The second approach aims at providing a language enriched with programming facilities so as to solve search problems in a way transparent to the user. This approach which led to the class of CLP(FD) languages is presented here.

For a different purpose, constraint satisfaction techniques have been embedded in LP to deal with real intervals. This corresponds to the class of CLP(Intervals) languages based on approximations of reals using real intervals.

3.3.1 CLP(FD)

Van Hentenryck and Dincbas [HD86] embedded constraint satisfaction techniques into logic programming by extending the concept of logical variable to the one of domain-variables which take their value in a finite discrete set of integers.

The key idea is to introduce the domain concept inside logic programming. This requires extending the unification procedure to the case of domain variables, thus making it possible to handle constraints using consistency techniques as inference rules. In particular the search techniques (forward checking, lookahead and partial lookahead) have been embedded into logic programming as inference rules [HD87]. The idea is that the way these techniques handle constraints can be applied locally to specific constraints, thus allowing for the most appropriate solving method. For example, the partial look-ahead inference rule deals efficiently with arithmetic expressions involving a large amount of variables. In practice this amounts to reasoning over domain bound variations. This has given birth to the first finite domain constraint logic programming language CHIP (Constraint Handling In Prolog [DSea88]). Constraints are arithmetic equations, inequalities and disequations over natural numbers as well as some symbolic constraints.

This "clever" manipulation of constraints which leads to efficient pruning with respect to one constraint follows the basic idea of earlier solvers for CSPs like REF-ARF [Fik70] and ALICE [Lau78]. REF is a nondeterministic programming language accepted by the problem solver ARF. The solver is based on the notions of node and arc consistency. In ALICE, the constraints are expressed in a mathematical language based on relation theory and some notions of graph theory. The searched objects are functions which should satisfy a set of constraints. The solver combines a depth-first search method with sophisticated constraint manipulation techniques and a set of powerful heuristics. The lack of flexibility of these seminal systems both in the language representation and the solving strategy motivated the design and implementation of CHIP.

The success of CHIP in the solving of a large class of combinatorial search problems like car-sequencing, warehouse location, investment planning, etc. [DSH88a] [DSH88b] [DHS+88] [Hen91] started the development of new finite domain CLP languages based on new features and implementations. The basic difference is that the user is not able any longer to specify how to use constraints unless they are user-defined constraints. Most of the systems solve the constraints using some local transformation rules based on consistency notions which are handled by a relaxation algorithm resembling AC-3. It uses a delay mechanism and suspension handling coroutines to wake the constraints which have to be reconsidered.

Later systems include ECL^{i}PS^e based on the notion of attributed variables [Hui90][Hol92] and a suspension mechanism which handles the delay and wakening of goals and constraints. It provides the features necessary to allow a user to develop his own constraint solver over a specific computation domain. cc(FD) [HSD93] is another successor of CHIP based on the AC-5 arc consistency algorithm. This language is defined as an instance of the cc framework[2] over finite domains. It adds to the finite domain library of CHIP a set of additional general-purpose combinators (such as cardinality, implication, constructive disjunction).

The early designers of CHIP also developed a new version CHIP V4 which includes a set of new global constraints [Bel90b] [BC94]. These constraints aim at reasoning globally over a set of constraints, versus local reasoning over one constraint. Recently some powerful techniques from operations research have been considered to increase the efficiency of the solving. From a practical point of view, they extend the application domain of the CHIP language to tackle efficiently graph and scheduling problems.

The finite domain library of CHIP has also given birth to a class of industrial languages like CHARME [OPL89], SNI-Prolog, Decision Power, ILOG solver [Pug92][CP94] among others.

3.3.2 CLP(Intervals)

This class of languages embeds the notion of domain with a different meaning. A domain specified by an interval does not represent a set of possible values a variable could take, but an approximation of a value. This research has been motivated by the errors resulting from finite precision arithmetic in computers. Each interval is marked by its lower and upper bounds which may or may not be included in the interval (open and closed intervals). This approach has been first implemented in Prolog from a functional viewpoint [Bun84]. It provides correct information about the range of the functions, but it prevents us from representing a logical real variable and from solving equations (eg. $[3.14, 3.142] = X + Y$ can not be solved).

Cleary [Cle87] introduced a relational arithmetic of real intervals into logic programming to avoid the weaknesses of the functional approach. The relational form of interval arithmetic is based first on the internal representation of reals as approximated intervals and second on the interpretation of arithmetic expressions as relations. This relational form can be nicely embedded into logic programming. Such a relation is specified as a subset of a Cartesian product of the real intervals involved in it. To make sure that the approximated intervals are the unique smallest ones which contain acceptable real values, Cleary makes use of projection functions and convex closure operations which allow the representation of each real interval appearing in a relation in terms of the other intervals which appear in the Cartesian product. The closure operations aim at guaranteeing that the computed intervals are convex that is they do not contain "holes". They constitute

[2] concurrent constraint framework, cf. *ask* & *tell* connectives

a second level of approximation. Indeed, some projection functions associated to the multiplication relation, for example, do not necessarily derive convex intervals. Thus the derived disjunctions of intervals are approximated by a closed one. This approach which does not allow "holes" in the intervals, might infer that some values in the intervals are inconsistent but are kept to avoid manipulating unions of intervals. Cleary proposed a solution to this problem, consisting in splitting the consistent intervals into sub_intervals and then checking whether some further restrictions can be deduced by performing nondeterministic computations over the disjunctive intervals.

A relaxation algorithm based on Waltz' filtering algorithm [Wal75] (or AC-3) processes a system of constraints by handling the various projection functions. It makes use of delay mechanisms to reconsider the relations whose Cartesian product has changed. The practical framework described by Clearly has been embedded in various languages referred to as CLP(Intervals). All of them are based on the relational form of interval arithmetic and the use of a relaxation algorithm to process a system of constraints. They do not handle the splitting of real intervals since it has been shown that handling disjunctions of intervals leads to a combinatorial explosion because of the large number of choice points which are generated once a disjunction is maintained and propagated.

A theoretical framework for the class of CLP(Intervals) languages has been described in [BMH94][Ben95]. It describes the key notion of approximation and the one of "narrowing operators" (cf. the projection functions) which derive the closest intervals from the previous ones so that the non relevant values are removed. The relaxation algorithm is referred to as the fixed point algorithm but provides the same constraint propagation and handling of the narrowing operators.

This class of CLP(Intervals) languages differs from that of CLP(FD) languages in the sense that an interval which is not reduced to one value might be a possible solution. This does not fit with CSP solving where a domain is a set of possible values and a solution should contain only variables instantiated to one domain value. The notion of approximated reals is very much related to the correctness issues and does not aim at solving a CSP.

Part II

The Language

4
Formal Framework

C'est au sommet de tes questions,
que tu trouveras la réponse.

This chapter describes the formal framework of a constraint logic programming language dealing with sets which range over a finite domain —*i.e.*, sets which belong to a powerset. The first step is the definition of the computation domain and syntax of the language that consists of the usual set operations and relational symbols ($\cup, \cap, \setminus, \subseteq$). The second step is the constraint solving part. The set satisfiability problem is $\mathcal{NP}$-complete and thus partial constraint solving is required (to the detriment of completeness but improving efficiency). The focus is on the definition of the constraint logic programming system which performs local consistency techniques over constraints of the language. The main idea is to specify each set domain by a set interval and to check the consistency of the constraints using set interval reasoning. In particular, it is described how the constraint domain of the system should be structured so as to deal with set intervals. This requires, among other things, to approximate the domain of a set expression (which might contain "holes") by a set interval and to define a set interval calculus. It is then shown how computations are performed over the constraint domain using a top-down execution model.

A constraint logic programming language with sets, set operations and relations is not expressive enough to tackle set based search problems. In particular optimization problems require the statement of an optimization function which necessarily deals with quantifiable, *i.e.* arithmetic, terms. To cope with this, an extension of the language is presented and consists in adding to the language syntax and to the constraint domain of the system a class of functions which map sets to integers (e.g. the set cardinality #, the set weight, etc.). These functions are called graduations when they map elements from a lattice (e.g. a powerset equipped with the operations $\cup, \cap$ and the partial ordering $\subseteq$) to the set of integers.

4.1 Basics of powerset lattices

Some definitions, properties and results on lattices are necessary to understand the main features of the formal language description. These can be found in [Bir67] [BM70] [Gea80]. The particular lattice we deal with is the powerset lattice. To give an intuitive idea of the subsequent use of these definitions, some examples relating to powerset lattices are given. Readers familiar with these notions can skip this section.

4.1.1 Lattices

Definition 6 *A* poset *(also known as* partially ordered set*) is a set S equipped with a binary relation $\preceq$ (formally a subset of $S \times S$) that satisfies the following laws:*

P1. Reflexivity	$\forall x, x \preceq x$	
P2. Antisymmetry	$(x \preceq y$ *and* $y \preceq x)$	$\Rightarrow (x = y)$
P3. Transitivity	$(x \preceq y$ *and* $y \preceq z)$	$\Rightarrow x \preceq z$

Example 7 Let S be a finite set and $\mathcal{P}(S)$ the set of all subsets of S or powerset of S. Then the set inclusion $\subseteq$ is easily seen to be a partial order on $\mathcal{P}(S)$. $\mathcal{P}(S)$ is a poset.

Definition 8 *Let S be a poset, X a subset of S and y an element of S. Then y is a* meet *or* greatest lower bound *or* glb *for X iff:*

y is a lower bound for X, i.e., if $x \in X$ then $y \preceq x$ and,
if z is any other lower bound for X then $z \preceq y$

The notation we use is $y = \bigwedge(X)$.

Definition 9 *Let S be a poset, $X \subseteq S$ and $y \in S$. Then y is a* join *or* least upper bound *or* lub *for X iff:*

y is an upper bound for X, i.e., if $x \in X$ then $y \succeq x$ and,
if z is any other upper bound for X then $z \succeq y$

The notation we use is $y = \bigvee(X)$.

Proposition 10 *Let S be a poset and X a subset. Then X can have at most one* meet *and at most one* join.

Proof By *P2*, meet and join are clearly unique whenever they exist. If a and b are two meets then we have on the one hand $a \preceq b$ and on the other hand $b \preceq a$. This infers $a = b$.□

The following property establishes a link between $\preceq$ and the pair $(\bigwedge, \bigvee)$ as actual meet and join.

Property 11 (Consistency property) *Let S be a poset. Then for all $x, y \in S$,*

$$x \preceq y \Leftrightarrow x = \bigwedge(\{x, y\})$$
$$x \preceq y \Leftrightarrow y = \bigvee(\{x, y\})$$

Definition 12 *A poset is a* lattice *iff every finite subset has a meet and a join.*

Corollary 13 *A poset S is a lattice iff every two elements have a meet and join.*

Example 14 The powerset $\mathcal{P}(X)$, is a lattice where the meet operator is the intersection $\cap$ and the join operator is the union $\cup$. Every two elements x, y of $\mathcal{P}(X)$ have a meet $x \cap y$ and a join $x \cup y$.

The partial order as set inclusion $\subseteq$ satisfies the consistency property:

$$x \cup y = y \Leftrightarrow x \subseteq y \Leftrightarrow x \cap y = x$$

This equivalence defines the correspondence between the relational definition of the structure in terms of properties of the partial order (existence of a glb and a lub) with the algebraic one (properties of the operations).

Definition 15 *A lattice L is* distributive *iff for every x, y and $z \in L$ we have:*

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$

Example 16 The powerset lattice is a distributive lattice.

4.1.2 Intervals as lattice subsets

Reasoning with and about intervals within a powerset lattice constitutes the core of our language. The following definitions and properties give the basic properties of intervals in lattices. An interval delimited by two elements x and y is specified by $[x, y]$.

Definition 17 *An* interval *of two arbitrary elements x, y in a lattice is the set $[x \wedge y, x \vee y]$.*

Definition 18 *A subset S of a lattice L is* convex *if $x, y \in S$ imply*

$$[x \wedge y, x \vee y] \subseteq S$$

Corollary 19 *A convex subset of a lattice is itself a lattice.*

Corollary 20 *A closed interval $[x \wedge y, x \vee y]$ is convex.*

Example 21 Let S be a subset of the powerset $\mathcal{P}\ (X)$. For every two elements $x, y \in S$ we have $[x \cap y, x \cup y] \subseteq S$. This interval is convex. Furthermore it is unique since the meet and join of x and y are unique.

Property 22 *The meet and join operators in a lattice are isotone (preserve the order):*

$$x \preceq y \Rightarrow x \wedge z \preceq y \wedge z$$
$$x \preceq y \Rightarrow x \vee z \preceq y \vee z$$

Example 23 This property is extremely useful when reasoning about intervals in a powerset lattice $\mathcal{P}\ (X)$. Consider the following inclusion relations between elements of $\mathcal{P}\ (X)$:

$$a \subseteq x \subseteq b \text{ and } c \subseteq y \subseteq d$$

x and y belong to the respective intervals $[a, b]$ and $[c, d]$. From property 22, we infer $a \cap c \subseteq x \cap y \subseteq b \cap d$ and dually for the union operation. So if x and y are only defined from the intervals they belong to, their union and intersection can be approximated by the new intervals $[a \cup c, b \cup d]$ and $[a \cap c, b \cap d]$.

4.1.3 Graduations

A graduation is a specific function which maps elements from a partially ordered set to the set of integers. For example, the powerset $\mathcal{P}(X)$ is graduated by the cardinality function. The following definitions give necessary conditions to consider graduations for a given set.

Definition 24 *A set S provided with an order relation $\preceq$ is graduated if there exists a function f from S to $\mathcal{Z}$ (positive and negative integers) which satisfies:*

$$x \prec y \Rightarrow f(x) < f(y) \quad (\prec \textit{ is a strict ordering, } < \textit{ the arithmetic inequality})$$
$$x \textit{ precedes } y \Rightarrow f(x) = f(y) + 1$$

An element x_i precedes an element x_{i+1} if in the chain of elements $x = x_0 \prec x_1 \prec ... \prec x_n = y$ in S there is no other element between them.
f is the graduation *of S.*

The existence of a graduation of a set which does not correspond to a chain (e.g. a set of set intervals) is guaranteed if the set is a lower semi-modular lattice.

Definition 25 *A lattice L is* lower semi-modular *if:*

$$x, y, z \in L \quad x \succ z \text{ and } y \succ z \Rightarrow (\exists t : t \succ x \text{ and } t \succ y)$$

Property 26 *The lattice of closed set intervals is a lower semi-modular lattice.*

Proof The semi-modularity of a lattice of set intervals derives directly from the existence of a lower and upper bound for any two intervals. Consider the strict orderings $[a_1, b_1] \supset z$ and $[a_2, b_2] \supset z$, z exists since the interval $[a_1 \cup a_2, b_1 \cap b_2]$ is one possible value for z. Then $t = [a_1 \cap a_2, b_1 \cup b_2]$ satisfies the condition: $t \supset [a_1, b_1]$ and $t \supset [a_2, b_2]$.$\square$

Consequently there exists a graduation for the lattice of closed intervals.

Property 27 *If there exists one graduation of a set, then there exists an infinite number of graduations of this set.*

4.2 Set intervals in CLP

Consider a set as an element of a powerset. Take the convex superset of this collection of sets (powerset). This convex part denotes a set interval. This concept of set interval is the means we will use to reason with and about sets in a Constraint Logic Programming (CLP) language[1]. On the one hand, the user manipulates sets in a logic-based language and on the other hand set interval calculus is performed to search for set values. The logic-based language is characterized by a set of predefined function and predicate symbols needed to deal with sets. This section describes the abstract syntax of the language and the algebraic structure of the system called the constraint domain. This is the structure over which set interval calculus is performed.

4.2.1 Abstract syntax and terminology

The syntax of the language comprises the set of predefined function and predicate symbols relative to sets, the set of constants, the variables, etc.

[1] A CLP language is a logic-based language parameterized by its computation domain and more generally by its constraint domain.

The alphabet The set of predefined function and predicate symbols necessary to reason with and about sets is the alphabet Σ_S:

$$\Sigma_S = \{\emptyset, \cup, \cap, \setminus, \subseteq, \in_{[a,b]}\}$$

The predicate symbol $\in_{[a,b]}$ applied to a variable s will be interpreted as the double ordering $a \subseteq s \subseteq b$.

Constants and terms The set of constants defines the domain of discourse of the language. It extends the Herbrand universe to provide the concept of set constant.

Definition 28 *The domain of discourse is the powerset*

$$D_S = \mathcal{P}(H_u) \textit{ where } H_u \textit{ refers to the Herbrand universe}$$

A set constant is any element from $\mathcal{P}(H_u)$ represented by the abstract syntax $\{e_1, ..., e_n\}$ where the e_i belong to H_u.

Definition 29 *A* set variable *is any variable taking its value in* $\mathcal{P}(H_u)$.

Definition 30 *A* set term *is defined by:*

(1) *any* set constant a *is a set term*
(2) *any* set variable s *is a set term*

Definition 31 *A* set expression S *of* $\mathcal{D_S}$ *where* S_1, S_2 *are set expressions is inductively defined by:*

$a \mid s \mid S_1 \cup S_2 \mid S_1 \cap S_2 \mid S_1 \setminus S_2$

Formulas and programs An atomic formula is a first-order atom (or atom) or a predefined constraint built from set terms, function and predicate symbols in Σ_S.

Definition 32 *An* atomic formula *is defined as follows:*

If p *is an n-ary predicate and* $t_1, ..., t_n$ *are terms, then* $p(t_1, ..., t_n)$ *is an atom.*

A program built from the logic-based language is based on definite clauses of the form:

$$(1)\quad A :- B_1, ..., B_n \quad \text{and} \quad (2) \quad :- G_1, ..., G_n$$

where A is an atom and the B_i, G_i are atoms or constraints. (1) is called a program clause and (2) a program goal. While atoms are not subject to a specific interpretation in the language, the predefined constraints characterize the language.

Notations Set variables will be represented by the letters x, y, z, s. Set constants will be represented by the letters a, b, c, d. Natural numbers will be represented by the letters m, n and integer variables by v, w. All these symbols can be subscripted.

4.2.2 Computation domain

The computation domain of the language is the powerset algebra $\mathcal{D}_S$ which interprets (over the domain of discourse D_S) the function symbols $\cup, \cap, \setminus$ belonging to Σ_S in their usual set theoretical sense (*i.e.*, $\emptyset$ is the empty set, $\setminus$ the set difference, etc.).

The interpreted set union and intersection symbols have the following algebraic properties:

C.	$x \cap y = y \cap x$	$x \cup y = y \cup x$	*commutativity*
As.	$(x \cap y) \cap z = x \cap (y \cap z)$	$(x \cup y) \cup z = x \cup (y \cup z)$	*associativity*
I.	$x \cap x = x$	$x \cup x = x$	*idempotence*
Ab.	$x \cap (x \cup y) = x$	$x \cup (x \cap y) = x$	*absorption*

4.2.3 Constraint domain

The constraint domain represents the structure of the system over which set interval calculus is performed. This structure is built from the computation domain equipped with the predicate symbols $\subseteq, \in_{[a,b]}$ belonging to Σ_S and interpreted as constraint relations. The predicate symbol $\subseteq$ is interpreted as the set inclusion and the predicate $\in_{[a,b]}$ is interpreted as the set domain constraint. This relation constrains a set variable to take its value in a specific domain. Since the main idea of the system is to perform set interval calculus, we must guarantee that the domain of any set is an interval.

The structure $[\mathcal{D}_{\mathcal{S}}, \subseteq]$ describes a powerset lattice with the partial order $\subseteq$. Any two of its elements a, b have a unique least upper bound $a \cup b$ and a unique greatest lower bound $a \cap b$ (cf. section 3.1.1.). The existence of limit elements for any set $\{a, b\}$ belonging to D_S allows us to define a notion of set domain as a convex subset of D_S, that is a set interval $[a \cap b, a \cup b]$.

Definition 33 *A* set interval domain *or* set domain *is a convex subset of D_S specified by $[a, b]$ such that $a \subseteq b$ and $a, b \in \mathcal{P}(H_u)$.*

Definition 34 *A set variable s is said to* range *over a set domain $[a, b]$ if and only if $s \in [a, b]$.*

The greatest lower bound a of the set domain contains the *definite* elements of s and the least upper bound b contains *possible* elements of s (comprising the definite ones).

Example 35 *The constraint $s \in [\{3, 1\}, \{3, 1, 5, 6\}]$ means that the elements $3, 1$ belong to s and that 5 and 6 are possible elements of s.*

Set intervals have been used so far to specify the domain of a set variable. Regarding set expressions, the domain of a union or intersection of sets is not a set interval because it is not a convex subset of D_S (e.g. $I = [\{1\}, \{1, 3\}] \cup [\{\}, \{2, 6\}]$, $\{1, 3\}, \{6\} \in I$ but $[\{\}, \{1, 3, 6\}] \not\subseteq I$). It is possible to maintain such disjunctions of domains during the computation, but this leads to a combinatorial explosion. This handling of "holes" can be avoided by considering the convex closure of a set expression domain. Consequently, the constraint domain of the system is defined as the powerset lattice over the convex parts of $\mathcal{P}(D_S)$ (convex subsets of D_S), equipped with a convex closure operation.

Definition 36 *The set of all convex parts of $\mathcal{P}(D_S)$ is a subset of $\mathcal{P}(D_S)$ ordered by set inclusion and designated by ΩD_S.*

Definition 37 *The* constraint domain *$\mathcal{CD}$ is the algebraic structure of the lattice ΩD_S of set intervals ordered by set inclusion such that:*

$$\mathcal{CD} = [\Omega D_S, \mathcal{D}_{\mathcal{S}}, \subseteq, \in_{[a,b]}]$$

The set equality can be derived from the double inclusion $x = y \Leftrightarrow x \subseteq y$ and $y \subseteq x$.

Convex closure operation. To ensure that any set domain is a set interval, we define a convex closure operation which associates to any element of $\mathcal{P}(D_S)$ its convex closure as being a set interval, element of ΩD_S.

Definition 38 *The* convex closure *operation* $\vec{conv} : \mathcal{P}(D_S) \rightarrow \Omega D_S$ *is such that* $\vec{conv} \; : x \rightarrow \overline{x}$ *satisfies:*

$$x = \{a_1, ..., a_n\} \rightarrow \overline{x} = [\bigcap_{a_i \in x} a_i, \bigcup_{a_i \in x} a_i]$$

Example 39 *The convex closure of the set* $\{\{3,2\}, \{3,4,1\}, \{3\}\}$ *belonging to* $\mathcal{P}(D_S)$ *is the set interval* $[\{3\}, \{1,2,3,4\}]$.

Property 40 *An element x of $\mathcal{P}(D_S)$ is convex under the above convex closure operation when x is equal to its "closure" $\overline{x}$.*

Corollary 41 *All singleton sets are convex.*

In the following, the operations $\bigcap_{a_i \in x} a_i$ and $\bigcup_{a_i \in x} a_i$ will be respectively written $glb(x)$ and $lub(x)$ which stand for greatest lower bound and least upper bound of x, respectively.

Property 42 *The operation* $\vec{conv}(x) = \overline{x} = [glb(x), lub(x)]$ *has the following* properties*:*

C1.	$x \subseteq \overline{x}$	*Extension*
C2.	$\overline{x} = \overline{\overline{x}}$	*Idempotence*
C3.	*If* $x \subseteq y$, *then* $\overline{x} \subseteq \overline{y}$	*Monotony*

If we consider the $\subseteq$ relation as a logical implication, the extension property *C1* can be interpreted by "any element of x belongs to $\overline{x}$ (thus to $glb(x)$) and any element definitely not in $\overline{x}$ (not in $lub(x)$) does not belong to x". This allows the set calculus to be performed in ΩD_S while ensuring that the computed solutions are valid in D_S. Property *C3* guarantees that the partial order $\subseteq$ is preserved in ΩD_S.

ΩD_S equipped with the operation $\vec{conv}$ allows us to define the constraint domain from an algebraic point of view, *i.e.*, from the properties of the union and intersection operations in ΩD_S.

Definition 43 *The constraint domain* $\mathcal{CD}$ *is a powerset lattice* $[\mathcal{D}_\mathcal{S}, \subseteq, \in_{[a,b]}]$ *with the family* ΩD_S *of set intervals that satisfies:*

P1. *Each* union *of elements of* ΩD_S *is also an element of* ΩD_S
P2. *Each* finite intersection *of elements of* ΩD_S *is also an element of* ΩD_S
P3. $\mathcal{P}(D_S)$ *and the empty set* $\{\}$ *are elements of* ΩD_S.

Properties *P1* and *P2* define the distributivity of $\cup$ and $\cap$ in ΩD_S. The conditions in *P3* define ΩD_S as a topology on $\mathcal{P}(D_S)$. It follows from *P2* and the first statement of *P3* ($\mathcal{P}(D_S) \in \Omega D_S$) that a convex closure operation satisfying *C1-C3* is defined in $\mathcal{CD}$. This operation is $\vec{conv}$. Because of *P1* and *P2* this operation satisfies

$$\overline{x \cup y} = \overline{x} \cup \overline{y} \text{ and } \overline{x \cap y} = \overline{x} \cap \overline{y}$$

Finally *P3* implies that $\overline{\emptyset} = \emptyset$.

4.2.4 Set interval calculus

In order to satisfy the properties *P1, P2* and *P3*, we define a set interval calculus within ΩD_S. This consists in deriving equality relations from the following ordering relations:

$$[a, b] \cup [c, d] \subseteq [a \cup c, b \cup d] \text{ and } [a, b] \cap [c, d] \subseteq [a \cap c, b \cap d]$$

This is achieved by making use of the convex closure operation. The resulting set interval calculus is described as follows:

$$\begin{array}{l} \overline{[a, b] \cup [c, d]} = [a \cup c, b \cup d] \\ \overline{[a, b] \cap [c, d]} = [a \cap c, b \cap d] \\ \overline{\mathcal{P}(D_s)} = \mathcal{P}(D_s) \text{ and } \overline{\emptyset} = \emptyset \end{array}$$

With regard to the set difference operation $[a, b] \setminus [c, d]$, its set theoretical definition is $x \setminus y = x \cap y'$ where y' is the complement of y. The complement of a set interval is characterized only by the fact that it does not contain the elements in the lower bound (e.g. c in this case). So the convex closure of a set interval difference is:

$$\overline{[a, b] \setminus [c, d]} = [a \setminus c, b \setminus c]$$

The consistency property $x \subseteq y \Leftrightarrow y = x \cup y$ and $x \subseteq y \Leftrightarrow x = x \cap y$ (cf. 3.1.1 property 11) establishes a link between $\subseteq$ and the set operations of a powerset lattice. This embeds the notions of right inclusion ($y = x \cup y$), which defines the consistency of y with respect to x, and the left inclusion ($x = x \cap y$). Intuitively the right inclusion aims at possibly adding elements to y and the left inclusion at possibly removing elements from x. Consequently, if a set interval $[a, b]$ specifies the set domain of a set variable x, the right inclusion is applied to a and the left inclusion to b. This is due to the fact that a contains elements which are already in x and b contains possible elements of x.

Definition 44 *Assuming that $[a, b]$, $[c, d]$ specify set domains, the consistency property in $\mathcal{CD}$ is defined by:*

$$[a, b] \subseteq [c, d] \quad \Leftrightarrow \quad b = b \cap d, c = c \cup a$$

This definition of consistency is fundamental from an operational point of view. It gives us the necessary conditions to be satisfied when checking and/or inferring consistency of the set inclusion constraint over set domain variables.

4.2.5 Graduations

The expressivity of the language can be increased if some "graded" functions are applied to set terms. A graded function maps a non quantifiable term to an integer value denoting a measure of the term. The set cardinality is one example of such a function. They allow the user to deal with optimization functions in a set-based language (e.g. minimizing the cardinality of a set). The constraint domain presented so far does not contain any such graded functions. In this subsection, we extend the alphabet of the language and the constraint domain of the system to deal with such functions. In lattice theory, a function which maps elements from a lattice structure (e.g. the constraint domain) to the set of integers, is called a graduation. Not all lattices can be equipped with graduations. One sufficient condition for this is that the lattice is lower semi-modular (cf. subsection 3.1.3). This is the case for $[\mathcal{D_S}, \subseteq]$ and for $[\Omega\mathcal{D_S}, \subseteq]$.

In order not to limit the extension of the language to the set cardinality function, the general case of an arbitrary graduation f is studied.

Definition 45 *A graduation f is a function from $[\mathcal{D_S}, \subseteq]$ to $\mathcal{Z}$ (set of positive and negative integers) which maps each element $x \in D_S$ to a unique m such that $f(x) = m$.*

The convex closure of a graduation f is required to deal with elements from ΩD_S. The closure function, written $\overline{f}$, maps elements from ΩD_S to a subset of the powerset $\mathcal{P}(\mathcal{Z})$ containing intervals of positive and negative integers. This subset is designated by $\Omega\mathcal{Z}$.

Example 46 *Let s be a set and $\#s$ its cardinality (a positive integer). Consider the constraint $s \in [\{\}, \{1, 2\}]$. The cardinality function $\#$ is approximated by $\overline{\#}$. Intuitively we have $\overline{\#}(s) = [0, 2]$.*

Definition 47 *Let $f : D_S \to \mathcal{Z}$. The function $\overline{f} : \Omega D_S \to \Omega\mathcal{Z}$ is derived from f as follows:*

$$\overline{f}([a, b]) = [f(a), f(b)]$$

Property 48 *If $x \in [a, b]$ then $f(x) \in \overline{f}([a, b])$.*

Proof. By definition f is a graduation. So if $a \subset x \subset b$ then we have $f(a) < f(x) < f(b)$. Consequently $f(x) \in [f(a), f(b)]$ which means $f(x) \in \overline{f}([a, b])$.$\square$

This property guarantees that the output of the function $\overline{f}$ applied to a set domain contains the actual graduation value of the concerned set variable.

4.2.6 Extended constraint domain

Graduations add expressive power to the language. They can be embedded as predefined symbols in the language, if the constraint domain is extended to deal with integer intervals and integer variables. The constraint domain associated with integer intervals is that of integer interval domains (subset of the standard constraint domain over finite integer domains). It is defined by the structure:

$$\mathcal{FD} = [\Omega\mathcal{Z}, (\mathcal{Z}, +), =, \neq, \geq, \in_{[m,n]}]$$

where the relation $\in_{[m,n]}$ is interpreted in $\Omega\mathcal{Z}$ as the integer domain constraint such that: $x \in_{[m,n]} [m, n]$ is equivalent to $m \leq x \leq n$. The other symbols are interpreted in their usual arithmetic sense. The extended constraint domain of our system should contain $\mathcal{FD}$.

Definition 49 *The extended constraint domain $\mathcal{CD}_e$ with graduations, is the structure:*

$$[\Omega D_S, \mathcal{D}_\mathcal{S}, f, \subseteq, \in_{[a,b]}] \cup \mathcal{FD}$$

$\mathcal{CD}_e$ interprets graduation symbols as unary set operations with respect to their intended meaning. For example the symbol $\#$ is interpreted as the set cardinality operation.

4.3 Execution model

The execution model is based on constraint solving in $\mathcal{CD}_e$. It is a top-down execution model which defines the operational semantics of the system. The model describes how the constraints are processed over $\mathcal{CD}_e$ and what they lead to. Since the set satisfiability problem is $\mathcal{NP}$-complete, it is a fortiori $\mathcal{NP}$-complete in $\mathcal{CD}_e$. For efficiency reasons, partial constraint solving is therefore required. The idea consists in transforming a system of constraints in $\mathcal{CD}_e$ as follows. Let each set variable range over a set domain. The transformation of a system of constraints in $\mathcal{CD}_e$ aims at removing some values of the set domains that can never be part of any feasible solution. This is achieved by making use of constraint satisfaction techniques.

A transformed system is commonly called a consistent system. One necessary condition for dealing with constraint satisfaction techniques is that each set variable ranges over a set domain. This section defines the various consistency notions for each predefined constraint in the system, gives the transformation rules used to infer consistency, and describes the operational semantics of the system as a transition system on states.

4.3.1 Definition of an admissible system of constraints

The set of predefined constraints in $\mathcal{CD}_e$ can contain any of the following:

- set domain constraints $s \in [a, b]$ where s is a set variable.
- set constraints $S \subseteq S_1$ where S, S_1 are set expressions (comprising constants, variables and possibly set operation symbols in $\{\cup, \cap, \setminus\}$).
- graduated constraints $\overline{f}(S) = [m, n]$ where $\overline{f}$ is any predefined graduation and $[m, n]$ any element in ΩZ (*i.e.*, an integer if $m = n$ or an integer domain).

Definition 50 *An* admissible system *of constraints in $\mathcal{CD}_e$ is a system of constraints such that every set variable s ranges over a set domain.*

4.3.2 From n-ary constraints to primitive ones

The predefined constraints might denote n-ary constraints like $s_1 \cup s_2 \subseteq s_3 \cap s_4$. The partial solving of constraints requires us to express each set variable in terms of the others. Since there is no inverse operation for $\cup, \cap, \setminus$ there is no way to move all the operation symbols on one side of the constraint predicate. So it is necessary to decompose n-ary constraints into primitive ones.

Consider the following set of basic set expressions $\{s \cap s_1, s \cup s_1, s \setminus s_1\}$. The proposed method consists in approximating each basic set expression by a new set variable with its appropriate domain. The resulting constraints are binary or unary ones called primitive constraints.

Definition 51 *A* primitive *constraint is (1) a predefined set constraint containing at most two set variables or, (2) a graduated constraint containing at most one set variable.*

In the former example the n-ary constraint is approximated by the system of constraints:

$$s_1 \cup s_2 = s_{12},\ s_3 \cap s_4 = s_{34},\ s_{12} \subseteq s_{34}$$

This approach is similar to the relational form of arithmetic constraints over real intervals introduced by Cleary [Cle87].

A relation denoting a basic set expression represent a subset of the Cartesian product of the set domains attached to each set variable. In order to deal with the consistency of these relations, we define projection functions which allow each set domain to be expressed in terms of the others. Consider a relation $r \subseteq [a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$. The set it denotes must belong to the domain ΩD_S over which the computations are performed. Since ΩD_S contains convex sets, each value of a projection function must be a convex set, that is a set interval. Consequently, to each projection function designated by ρ_i we associate its closure $\overline{\rho_i}$. The closure is derived from ρ_i by making use of the closure operator defined above which satisfies:

$$\overline{\rho_i} = c\vec{on}v(\rho_i)$$

$\overline{\rho_i}$ represents the approximation of this projection of the relational form r on the s_i-axis.

Definition 52 *The i-th projection function $\overline{\rho_i}$ of a relation r denoting a set expression is the mapping :*

$\overline{\rho_i} = c\vec{on}v\{s_i \in [a_i, b_i] \mid \exists (s_j, s_k) \in [a_j, b_j] \times [a_k, b_k]$ *such that* $j, k \neq i : \ (s_i, s_j, s_k) \in r\}$

These relational forms of set expressions are not visible to the user but they are necessary to define the consistency of an n-ary constraint.

4.3.3 Consistency notions

The consistency notions provide necessary conditions to ensure the partial satisfaction of primitive constraints. The standard notions of consistency applied to integer domains state conditions that must be satisfied by each element belonging to a variable domain. This approach is not useful to us since set domains specified by set intervals can contain an exponential number of elements (in the size of the powerset described by the domain bounds). Instead, we derive conditions that must be satisfied only by the domain bounds. These conditions guarantee that any relation which does not hold for the bounds of the variable domains will not hold for any element between these bounds. Consider a set variable s. The lower and upper bounds of the domain of s will be respectively defined by the functions $glb(s)$ and $lub(s)$. The upper letters S, S_1 denote set expressions.

Preliminary definitions With regard to the consistency properties of the set inclusion constraint, the concepts of lower and upper orderings have been informally introduced. Their formal definitions are given here since they will be of much use in the subsequent definitions. Assume the following notations: $\subseteq_L$ for the lower ordering and $\subseteq_U$ for the upper ordering.

Definition 53 *Let a, b denote ground sets. The lower ordering is the relation:*

$$a \subseteq_L b \Leftrightarrow \forall x \in a,\ x \in b$$

Definition 54 *Let a, b denote ground sets. The upper ordering is the relation:*

$$a \subseteq_U b \Leftrightarrow \forall x \notin b,\ x \notin a$$

These preliminary definitions allow us to define the consistency notions for primitive constraints.

Definition 55 *Let $s \subseteq s_1$ be a primitive set constraint. We say that this constraint is* consistent *if and only if:*

SC1. $glb(s) \subseteq_L glb(s_1)$ *and*
SC2. $lub(s) \subseteq_U lub(s_1)$.

The consistency of a primitive set constraint is equivalent to the standard notion of arc-consistency (*i.e.*, interval consistency is equivalent to domain consistency). Correspondingly, if a set constraint is an unary constraint, its consistency is equivalent to node consistency.

Property 56 *A primitive set constraint is consistent if an only if it is arc-consistent.*

Proof. This property holds because the operations $\cup$ and $\cap$ are isotone. The constraint $s \in [a,b]$ is equivalent to $\forall e_s \in [a,b]$ we might have $s = a \cup e_s$. The isotony of $\cup$ means that $a \subseteq e_s \subseteq b \Rightarrow a \subseteq e_s \cup a \subseteq b$ (since $a \subseteq b$).

Assume the domain constraints $s \in [a,b],\ s_1 \in [c,d]$. The set constraint $s \subseteq s_1$ is consistent iff:

$$
\begin{array}{lcl}
a \subseteq_L c \text{ and } b \subseteq_U d & \Leftrightarrow & \forall e_s \in [a,b]\ a \cup e_s \subseteq_L c \cup e_s \text{ and } b \cup e_s \subseteq_U d \cup e_s \\
 & & \forall e_s \in [a,b],\ \exists e_{s1} \in [c,d], e_{s1} = c \cup e_s \text{ such that} \\
 & & e_s \subseteq e_{s_1} \\
 & & s \subseteq s_1 \text{ is arc-consistent.}
\end{array}
$$

$\square$

Definition 57 *A primitive graduated constraint* $\overline{f}(s) = [m,n]$ *is* consistent *iff:*

SC3. $\overline{f}(s) = \overline{f}(s) \cap [m,n]$

The consistency of the relational forms of basic set expressions is defined through the consistency of the projection functions. Since the set domain of a basic set expression is approximated it is clear that we can not get the equivalent of arc-consistency. Some elements in the resulting set interval are meant to fulfill "holes" and are not expected to be part of any feasible solution.

Theorem 58 *A* relation r *denoting the relational form of a basic set expression is* consistent *if and only if each of the projection functions* $\overline{\rho_i}$ *describing* r *is consistent.*

Definition 59 *A projection function* $\overline{\rho_i}$ *associated to the relation* $r \subseteq \Pi_{j\in\{1,\ldots,3\}}[a_j, b_j]$ *is consistent iff:*

SC4. $glb(\overline{\rho_i}) \subseteq_L a_i$ *and* $b_i \subseteq_U lub(\overline{\rho_i})$

4.3.4 Inference rules

The consistency notions define conditions to be satisfied by set domain bounds so that a set constraint is consistent. If such conditions are not satisfied this means that elements in the domain are irrelevant. Consistency can be inferred by moving such elements "out of the boundaries of the domain" which means pruning the bounds of the domain. The essential point is that a refinement of both bounds allows us to prune a domain. Reducing the set of possible values a set could take can be achieved either by extending the collection of *definitive* elements of a set *i.e.*, satisfying the lower ordering, or by reducing the collection of *possible* elements *i.e.*, satisfying the upper ordering. Both computations are deterministic and are derived from the consistency notions.

4.3.4.1 For set constraints

Consider the constraint $s \subseteq s_1$ such that $s \in [a,b], s_1 \in [c,d]$. Inferring its consistency by means of a domain bound reasoning amounts to satisfying the lower ordering by possibly extending the lower bound of the domain of the set variable s_1 and satisfying the upper ordering by possibly reducing the upper bound of the domain of s. This is depicted by the following inference rule:

$$\text{I1.} \quad \frac{b' = b \cap d \;, \qquad c' = c \cup a}{\{s \in [a,b],\; s_1 \in [c,d],\; s \subseteq s_1\} \longmapsto \{s \in [a,b'] \;, s_1 \in [c',d],\; s \subseteq s_1\}}$$

When s, s_1 denote set expressions, the relational forms are created and the following additional inference rule is necessary to deal with the projection functions. For each projection function $\overline{\rho_i}$ describing the domain of an s_i appearing in a set expression, we have:

$$\text{I2.} \quad \frac{a_i' = a_i \cup c \;, \qquad b_i' = b_i \cap d}{\{\; s_i \in [a_i, b_i],\; \overline{\rho_i} = [c,d] \;\} \longmapsto \{\; s_i \in [a_i', b_i']\}}$$

Two additional inference rules describe the cases where the set domain of a set is reduced to one value or is inconsistent:

$$\text{I3.} \quad \frac{a = b}{\{\; s_i \in [a,b] \;\} \longmapsto \{\; s = a\}} \qquad\qquad \text{I3'.} \quad \frac{a \supset b}{\{\; s_i \in [a,b] \;\} \longmapsto fail}$$

4.3.4.2 For primitive graduated constraints.

The constraint $\overline{f}(s) = [m,n]$ such that $s \in [a,b]$ describes a mapping from an element belonging to a partially ordered set to an element belonging to a totally ordered set. Consequently, it might occur that two distinct elements in $[a,b]$ have the same valuation in $[m,n]$. This implies that inferring the consistency of this constraint might require refining $[a,b]$ only if a single element in $[a,b]$ satisfies the constraint. If this element exists, it corresponds necessarily to one of the domain bounds since they are uniquely defined and are strict subset (or superset), of any element in the domain. Thus, the value of the graduation mapped onto them can not be shared. The inference mechanism is depicted by the following rules:

$$\text{I4.} \quad \frac{[m',n'] = [m,n] \cap \overline{f}\;(s)}{\{\; s \in [a,b] \;, \overline{f}\;(s) = [m,n]\} \longmapsto \{s \in [a,b] \;, \overline{f}\;(s) = [m',n'] \;\}}$$

$$\text{I5.} \quad \frac{n = \overline{f}\;(a)}{\{\; s \in [a,b] \;, \overline{f}\;(s) = [m,n]\} \longmapsto \{s = a \;\}}$$

$$\text{I6.} \quad \frac{m = \overline{f}\;(b)}{\{\; s \in [a,b] \;, \overline{f}\;(s) = [m,n]\} \longmapsto \{s = b \;\}}$$

4.3.5 Operational semantics

The inference rules described so far can be applied to individual constraints. The operational semantics shows how to check and infer the consistency of a system of constraints. This system should correspond to an admissible system of constraints. The consistency of such a system results from the consistency of each constraint appearing in it. The operational semantics is based on one non deterministic transition rule which takes as input a goal comprising a collection of (1) set domain constraints A, (2) other constraints C, (3) two sets of atoms G and B and (4) one clause among the possible ones in the program whose head can be unified with the leftmost atom in G. The leftmost atom in G is marked out by $\uparrow G$, and the remainder of G by $\downarrow G$. This rule returns a new goal to be solved such that the set of constraints is consistent and possibly simplified. It is depicted in the following figure. The notation $\diamond$ is used to distinguish the sets of atoms from the sets of constraints.

from $\leftarrow A, C \diamond G$ and $a \leftarrow C_1 \diamond B$
infer $\leftarrow A_1, C_2 \diamond (\downarrow G \cup B)$
if $\{\ A, \{\uparrow G = a\} \cup C \cup C_1\} \longmapsto \{\ A_1, C_2\ \}$

Figure 4.1 Derivation rule of the operational semantics

The crucial point lies in the inference rule defined in the **if** statement. The inference rules defined so far deal with one constraint. From inferring of the consistency of one constraint, we move to inferring the consistency of a collection of constraints. At the same time, this inference rule possibly transforms the set of domain constraints and the set of the other constraints. The reason is that the consistency of some constraints might result from the requirements for domain refinements and thus a replacement of the previous set domain constraints (cf. I1, I2) and additionally some constraints might be simplified which leads to a transformation of the set of other constraints (cf. I3, I5, I6). This inference rule corresponds to a set of simple rules which describe the process in more detail.

The process amounts to considering a transition system on states where each state contains the new constraints as yet unconsidered, the (set, integer) domain constraints and the constraints which have already been checked out. One state is specified by a tuple $\langle C, A_s \cup A_i, S\rangle$ containing the following collections of constraints:

A set of as yet not considered constraints designated by C,
A set of set domain constraints designated by A_s,
A set of integer domain constraints A_i,
A set of consistent constraints S.

A_s, A_i and S are usually referred to as the constraint store. When A_s and A_i do not need to be distinguished their union is denoted A. The initial state of the transition system is $\langle C, \emptyset, \emptyset \rangle$ where all the constraints need to be checked.

The inference rule in the **if** statement contains different configurations of state transition. For example, one transition might be that the consistency of one constraint is inferred without any requirement for domain modification, or that it requires domain refinements which leads to the inconsistency of some already stored constraints. The following set of transition rules corresponds to the various possible transformations which are derived when checking or inferring consistency of one constraint in conjunction with the constraint store. The first two transition rules deal with consistency checking and the last two with the consistency inference.

$$\text{T1.} \quad \langle C \cup c, A_s, S \rangle \longrightarrow_c \quad \langle C, A_s, S \cup c \rangle$$

if c is consistent in conjunction with the set A_s and consequently with the constraint store. c is then added to the set of consistent constraints S.

$$\text{T2.} \quad \langle C, A, S \rangle \longrightarrow_c \quad fail$$

if at least one set domain or integer domain constraint in A is inconsistent. This transition is derived if the inference rule I3' succeeds over at least one set domain constraint. A similar inference rule for the case of integer domains is quite straightforward and corresponds to the case where $x \in [m, n]$ and $n > m$.

$$\text{T3.} \quad \langle C \cup c, A_s, S \rangle \longrightarrow_i \quad \langle C, A'_s, S \cup c \rangle$$

if the consistency of c is inferred by requiring a pruning of some set domains thus requiring to modify the set of set domain constraints A_s. This transition is derived if any of the inference rules I1, I2 and I3 is successfully applied.

$$\text{T4.} \quad \langle C \cup c, A_s \cup A_i, S \rangle \longrightarrow_i \quad \langle C, A'_s \cup A'_i, S \cup c \rangle$$

if the consistency of c is inferred by requiring a pruning of some integer domain (I4) and possibly some set domain (I5, I6). Consequently the sets A_s, A_i might get modified.

Each derivation rule takes an element from C and moves it to S. So the final state of the transition system is either *fail* or $\langle \emptyset, A', S' \rangle$.

Theorem 60 *A system of constraints S is consistent if and only if all the domain constraints that it contains are consistent.*

Proof This follows simply from the various inference rules. Inferring the consistency of a system amounts to considering the consistency of each constraint in conjunction with the already consistent ones. The system is detected inconsistent if and only if the inference rule I3' is successfully applied.$\square$

4.3.5.1 Satisfiability issue

Ensuring the satisfiability of a consistent system requires guaranteeing that a solution exists. This is in not possible when an n-ary set constraint happens to belong to the system since we work on domain approximations. But whenever dealing with unary and binary set constraints, property *56* (cf. equivalence between set constraint consistency and the usual consistency notions) guarantees a solution. The lower or upper bounds of the set domains will always be possible values for the sets. With respect to graduated constraints, consistency does not guarantee satisfiability since a consistent graduated constraint $\overline{f}\ (s) = m$ does not guarantee that some elements of the domain of s might satisfy the constraint.

Theorem 61 *A system of set constraints containing only unary and binary set constraints is satisfiable if and only if it is consistent.*

Proof. This follows simply from the property *56* which holds thanks to the monotony of the operations $\cup, \cap$.$\square$

5
Practical Framework

Le mot est créateur,
car il concentre tout, il centre.
Le mot construit.
Ce n'est pas sans raison que telle pierre
s'imbrique dans telle autre.
Autrement, ce que tu construis s'écroulerait.

This section describes the Conjunto[1] language, a constraint logic programming language designed and implemented to reason with and about sets ranging over a set domain. Its design is based on the notion of set defined as an individual element from a subset of a powerset universe. The functionalities of Conjunto (apart from those of a logic-based language) are set operations and relations from set theory together with some graduations which provide set measures like cardinality, weight, etc. We describe how these graduations can be reconsidered so as to map set domains to subsets of the natural numbers (finite domains).

The implementation of Conjunto is concerned with the way set calculus is achieved in algorithmic terms. Searching for a complete solution is an intractable problem since set satisfiability is an $\mathcal{NP}$-complete problem. The basic principle of the Conjunto solver is to check and infer a coherent system of set constraints which guarantees that set values which have been removed from the set domains can never be part of any feasible solution. This is achieved by adapting local consistency techniques to a domain bound reasoning. Particular attention is given to the description of the local transformation rules which perform domain refinements to infer local consistency of individual constraints. We then describe how the solver which infers/checks the consistency of a system of constraints, handles the calls to these rules by making use of delay mechanisms. Their adequacy to establish a dynamic cooperation between two solvers (Conjunto solver and finite domain solver) is illustrated by the handling of graduated constraints in conjunction with other constraints.

[1]Conjunto means "set" in Spanish

5.1 Design of Conjunto

This section describes the functionalities of the Conjunto language. We omit a detailed description of the traditional predicates and functions on Prolog terms [CKC83].

5.1.1 Syntax

The Conjunto language is a logic-based programming language with the alphabet of a Prolog language (constants, predicates, functions, connectives, etc). It is characterized by a signature Σ which contains the following set of predefined function and predicate symbols in their concrete syntax:

- the constant `{}`.
- the binary set predicate symbols {`‘<`, `‘<>`, `‘::`, `#`, `weight`} and arithmetic predicate symbols $\{=, \geq, \neq\}$.
- the binary set function symbols {`\/`, `/\`, `\`} and the arithmetic sum symbol `+`.

A Conjunto atomic formula is a first-order atom (referred to as atom) or any atomic formula referred to as primitive constraint built from variables, function and predicate symbols in Σ.

The language is based on definite clauses of the form:

$$(1) \quad a :- b_1, ..., b_n \quad \text{and} \quad (2) \quad :- g_1, ..., g_n$$

where a is an atom and the b_i, g_i are atoms or constraints. While atoms are not subject to a specific interpretation in the language, the constraints constitute the core functionalities of the language and are characterized by a specific terminology and semantics.

5.1.2 Terminology and semantics

The main objective of Conjunto is to perform set calculus over sets defined as elements from a powerset domain. Some constraints like set cardinality or set weight require us to deal also with finite domains, that is integers and arithmetic constraints.

Definition 62 *The computation domain is the set* $\mathcal{D} = \mathcal{P}(H_U) \cup H_U$ *where* $\mathcal{P}(H_U)$ *is the powerset of the Herbrand universe.*

5.1.2.1 Terminology

The terminology gives names to the predicate and function symbols in Σ and defines the notions of set domains and set terms necessary to reason with and about sets in $\mathcal{D}$.

The symbols in {`‘<`, `‘<>`, `‘::`, `#`, `weight`} refer respectively to the set inclusion constraint predicate, the set disjointness constraint predicate, the set domain constraint, the set cardinality constraint predicate and the weight constraint predicate. The symbols in {`\/`, `/\`, `\`} represent the concrete syntax of the set operations $\cup, \cap, \setminus$. The other symbols in Σ refer simply to the arithmetic operations they denote.

Definition 63 *A* ground set *is an element of* $\mathcal{P}(H_U)$ *which represents a finite set of Herbrand terms delimited by the characters* { *and* }.

Example 64 `{2,3,f(f(u,o))}` is a ground set.

Definition 65 *A* set domain *is a convex set of ground sets semantically equivalent to a set interval. It is denoted by* $[a, b]$ *where* a *and* b *are ground sets such that* $a \subseteq b$.

Definition 66 *A* weighted set domain *is a specific set domain where each element of the set domain bounds has the syntax* (e, m) *such that* e *is a Herbrand term and* m *is an integer.*

Definition 67 *A* set variable *is a logical variable whose value lies in a set or weighted set domain. Its syntax is* s = s{[a,b]}.

Example 68 `S= S{[{(a,1)},{(a,1),(c,2),(d,2)}]}` is a set variable whose weighted set domain is the set interval `[{(a,1)},{(a,1),(c,2),(d,2)}]`.

Definition 69 *A* set term *is a (1) a ground set, or (2) a set variable.*

Definition 70 *A* set expression s *is inductively defined by:*

$s ::= t_s \mid s_1$ `/\` $s_2 \mid s_1$ `\/` $s_2 \mid s_1$ `\` s_2

where s_1, s_2 *are set expressions, and* t_s *a set term.*

Similarly, variables denoting integers will take their value in a finite set of integers (finite domain). In Conjunto these domains are approximated by integer interval domains. An *integer interval domain* is the convex closure of a finite set of integers and will be simply referred to as an integer interval.

Definition 71 *An* integer variable *is a logical variable whose value lies in an integer interval.*

5.1.2.2 Semantics

The interpretation of the elements of Σ in $\mathcal{D}$ is given by distinguishing set constraints from graduated constraints.

Notation. Conjunto's predicate and function symbols are written in a bold font. Set variables are denoted s, v, w, set expressions t, integer variables are denoted x, y, z, ground sets a, b, c, d, integers m, n. These symbols may be subscripted.

A *primitive set constraint* is one of the following constraints:

- s `` `:: `` $[a, b]$ is semantically equivalent to $a \subseteq s \subseteq b$[2]
- s `` `< `` s_1 is equivalent to the set inclusion relation $s \subseteq s_1$.
- s `` `<> `` s_1 is equivalent to the empty intersection of the two sets s, s_1.

Remark The set disjointness constraint `` `<> `` which was not included in the formal part has been embedded as a primitive constraint in Conjunto mainly for practical reasons. Since the disjointness of two sets appears in almost all set based problems, it is simpler to use a specific syntax and more efficient to handle it as a primitive constraint.

A *primitive graduated constraints* is one of the following:

- `#`(s, x) is equivalent to the arithmetic equality $\#s = x$ where $\#s$ is the standard cardinality function of set theory.
- `weight`(s, x) is semantically equivalent to the arithmetic operation $\sum_i m_i = x$ such that $(e_i, m_i) \in s$.

[2]cf. the $\in_{[a,b]_{a \subseteq b}}$ predicate in the formal part.

The function symbols `\/`, `/\`, `\` are interpreted as the set operations $\cup, \cap, \setminus$, respectively, in their usual set theoretical sense. The set difference is a complementary difference (e.g. $s \setminus s_1 = \{x \in s \mid x \notin s_1\}$).

Definition 72 *The constraint system of a* Conjunto program *is an admissible system*[3] *of set constraints and graduated constraints where every set variable is constrained by a set domain constraint.*

In this admissible system of constraints the searched objects are the sets. The integer variables are not part of the initialization of the search space which is attached to the system. They constitute essentially a means to get to the final solution. This is described in the following corollary.

Corollary 73 *An admissible system of set and graduated constraints is a* set domain *constraint satisfaction problem* i.e., *a constraint satisfaction problem where the initial search space is defined by the set domains attached to the set variables.*

5.1.3 Constraint solving

The constraint solving in Conjunto focuses on efficiency rather than on completeness. Since the set satisfiability problem is $\mathcal{NP}$-complete, partial constraint solving is required. The Conjunto solver aims at checking and inferring the consistency of an admissible system of constraints. This is achieved by:

- applying some local transformation rules, which allow the consistency of one constraint to be checked/inferred, using a top-down search strategy,
- delaying consistent constraints which are not completely solved.

The Conjunto solver considers one constraint at a time and checks/infers its consistency in conjunction with the set of delayed constraints. This process might require the consistency of some delayed constraints to be reconsidered. These constraints are woken using a data driven mechanism based on suspension handling mechanisms.

The solver acts like a transition system on states. One state is denoted by a tuple of as yet unconsidered constraints together with a constraint store containing the delayed constraints. Each newly consistent constraint is added to the constraint store. The final state of the program is achieved when all atoms appearing in a goal clause have been checked and when no further domain refinement is

[3] cf. definition in the formal part 3.3.1

required. This state is either denoted by "fail" when some constraints have been marked inconsistent or it contains a set of delayed constraints together with the set variables and their associated domains.

Example 74 *The goal:*

```
:- S `:: [{1},{1,2,3,4}], S1 `:: [{3},{1, 2, 3}], S `< S1.
```

produces the refined domains:

```
S = S{[{1},{1,2,3}]}  S1 = S1{[{1,3},{1,2,3}]}
```

and the delayed goal: `S `< S1`

Example 75 *The goal:*

```
:- S `:: [{1},{1,2,3,4}], #(S,1).
```

produces the instantiation `S = {1}` *and no delayed goal since the initial goal is completely solved.*

5.1.4 Programming facilities

One of the application domains we have investigated using Conjunto is the modelling and solving of set based combinatorial problems. To allow the user to state short and concise programs, some programming facilities have been added to the initial set of primitive constraints. They consist of a collection of constraints defined from the primitive ones, some predicates necessary to access information related to the variable domains, and a built-in set labelling procedure. The most important ones are presented below, others are given in the annexe A.

5.1.4.1 Set constraints

The set equality t `` `= `` t_1 requires two set expressions to be equal. This constraint is simply derived from a double set inclusion: t `` `< `` t_1, t_1 `` `< `` t and is handled as such.

The membership and nonmembership e `in` s , e `notin` s are handled in a passive way in the sense that they are considered once e is ground. They are respectively defined in terms of set inclusion and set disjointness constraints if e is ground, and delayed otherwise: $\{e\}$ `` `< `` s and $\{e\}$ `` `<> `` s.

The global union `all_union`$([s_1, ..., s_n], s)$ requires the union of all the set terms in $[s_1, ..., s_n]$ to be equal to s. In case s is a free variable, it becomes a set variable and its domain is the union of the set domains or set values attached to the set terms. It is defined by means of pairwise unions. The handling of this constraint does not perform a global reasoning over the s_i but amounts to dealing with a collection of set equality constraints over a set variable and the union of two set variables. Even though this process is not visible to the user, the set equality constraints which are not completely solved appear in the set of delayed goals.

Example 76 *The goal:*

```
:- [S1,S2,S3] `:: [{},{a,b,c}], all_union([S1,S2,S3], {a,b})
```

produces the refined domains:

```
S1 = S1{[{},{a,b}]}, S2 = S2{[{},{a,b}]}, S3 = S3{[{},{a,b}]}
```

and the set of delayed goals:

`S1S2 \/ S3 `= {a,b}` *and* `S1 \/ S2 `= S1S2{[{},{a,b}]}`

The global disjointness `all_disjoint`$([s_1, ..., s_n])$ requires all the set terms in $[s_1, ..., s_n]$ to be pairwise disjoint. It is defined by means of disjointness constraints over every couple of s_i. It is handled in a way similar to the global union constraint.

5.1.4.2 Set domain access

Set domains are represented as abstract data types, and the users are not supposed to access them directly. So two predicates are provided to allow operations on set domains : `glb`(s, s_{glb}) and `lub`(s, s_{lub}). If s denotes a set variable, each term is respectively assigned the value of the domain's lower and upper bound. Otherwise it fails.

5.1.4.3 Set labelling

Assigning a value to a set variable is a nondeterministic problem which can be tackled by different labelling strategies. Since the Conjunto solver uses partial constraint solving, an adequate strategy should aim at making an active use of the constraints in the constraint store. On the one hand, a procedure which would consist in instantiating a set by directly selecting an element from the set domain makes a passive use of the constraints whose consistency is only partial. In the worst case this process might require considering all the elements belonging to a set domain even if some of them are irrelevant. On the other hand refining a set domain by adding one by one elements to the lower bound of the domain is more likely to minimize the possible choices to be made. The `refine` predicate embedded in Conjunto behaves as follows:

`refine`(s) labels s, if s is a set variable. If there are several instances of s, it creates choice points. If s is a ground set, nothing happens. If not, the following actions are performed recursively until the set gets instantiated: (1) select an element e from the ground set $lub(s) \setminus glb(s)$, (2) add the membership constraint e `in` s to the program. This added constraint is handled by the solver which checks its consistency in conjunction with the actual constraint store. In case of failure the program backtracks and (3) the nonmembership constraint is added (successfully) to the program so as to remove the irrelevant value e from the domain. The points (2) and (3) correspond to the disjunctive set of constraints:

```
( e in S ; e notin S)
```

Example 77 *Consider the goal:*

```
:- S `:: [{},{1,2,3}], refine(S).
```

The search tree generated during the labelling procedure and covered using a depth first search strategy is described in figure 5.1.

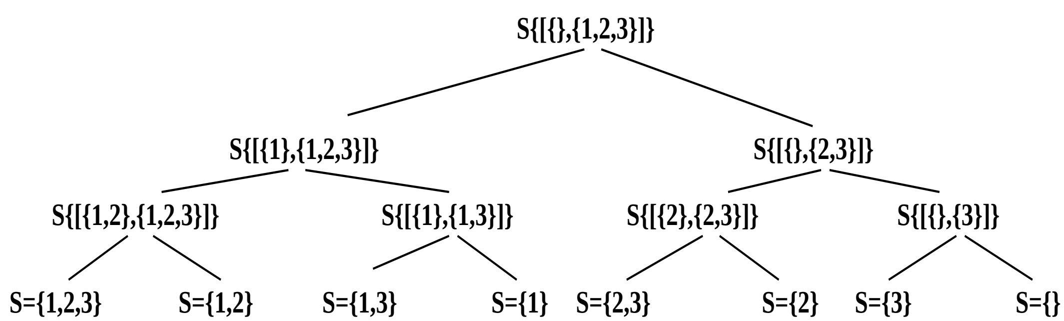

Figure 5.1 Example: search tree of the predefined labelling procedure

The strategy, which consists in adding membership constraints to the program, aims in particular at making an active use of those graduated constraints whose consistency is only partial.

Example 78 *Consider the goal:*

```
:- S `:: [{},{1,2,3}], #(S,1), refine(S).
```

The irrelevant branches of the search tree are cut in an a priori way i.e.*, no useless choice point is created. The search tree generated during the solving of this goal is depicted in figure 5.2.*

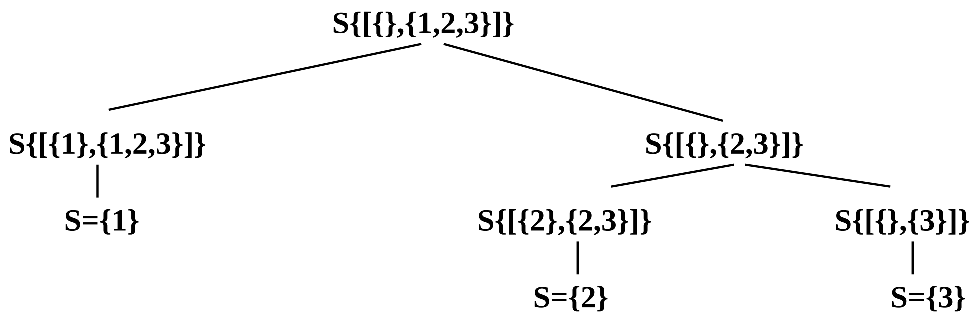

Figure 5.2 Example: cutting branches of the search tree

5.1.4.4 Optimization predicates

The notion of optimization is common in problem solving. It aims at minimizing or maximizing a cost function which denotes a specific arithmetic expression. The notion of cost defines a kind of measure or quantification applied to some terms. A set can not denote a quantity and is not measurable. Only its possible graduations are. Thus there are no specific optimization predicates for sets. Existing predicates embedded in a finite domains solver (e.g. for a branch and bound search) can be directly applied to expressions over integer intervals occurring in graduated constraints. For example, minimizing a set cardinality acts over a set through the link existing between a set variable and its cardinality.

5.1.4.5 Relations and constraints

When dealing with sets, it sounds quite natural to deal with relations and functions as well. Functions are more restrictive than relations since they constrain each element from its DS-domain[4] to have exactly one image. Providing relations at the language level extends the expressive power of the language when dealing

[4] DS-domain stands here for departure set

for example with circuit problems and matching problems originating from Operations research. In relation theory [Fra86], a relation $\mathcal{R}$ is represented as a set of ordered pairs (x_i, y_j) such that x_i belongs to the DS-domain d of $\mathcal{R}$ and y_j to its AS-range[5] a. In other words, a relation $\mathcal{R}$ on two ground sets d and a is a subset of the Cartesian product $d \times a$. Keeping this representation to deal with relations as specific set terms containing pairs of elements can be very costly in memory. Indeed, the statement of the Cartesian product referring to a relation requires us to consider explicitly a huge set of pairs. This is very inconvenient. Instead, a relation in Conjunto is represented as a specific data structure which is characterized by two ground sets (DS-domain and AS-range) and a list containing the successor sets attached to each element of DS-domain [Ger93a] [Ger93b]. Considering one successor set per element splits the domain of a relation into a collection of set domains. The resulting value of a relation is clearly the union of the successor sets. This approach is close to the one introduced in the seminal work ALICE [Lau78] which dealt essentially with functions. However in ALICE there is no explicit notion of set domain.

Definition 79 *Let a relation be $r \subseteq d \times a$. The successor set s of an element $x \in d$ is the set $s = \{y \in a \mid (x, y) \in r\}$.*

Definition 80 *A relation variable r is a logical variable whose value is a compound term* `birel`(l, d, a) *such that* `birel` *is a functor of arity three, l is a list of $\#d$ set variables s_i such that s_i* `` `:: [{},`` a`]` *and d, a are two ground sets.*

This compound term is associated to a free variable by means of the predicate r `bin_r` d `-->` a.

Example 81 *The goal:*

```
:- R bin_r {1,2} --> {a,b,c}.
```

creates the term:

```
R = birel([Set1{[{},{a,b,c}]}, Set2{[{},{a,b,c}]}], {1,2}, {a,b,c})
```

The definition of constraints applied to relation variables abstracts from stating directly constraints over the set DS-domain and AS-range or over the successor sets. The following constraints have been embedded in Conjunto:

[5] AS-range stands here for arrival set

- (i, j) `in_r` r, (i, j) `notin_r` r which adds or retrieves pairs to the relation
- `funct`(r) which constrains a relation to be a function,
- `inj`(r) which constrains a relation to be an injective function,
- `surj`(r) which constrains a relation to be an surjective function,
- `bij`(r) which constrains a relation to be an bijective function.

The schema of these constraints is directly derived from their usual interpretation issued from relation theory [Fra86]. They are represented below using the the mathematical cardinality operation $\#$, the usual set operation symbols $(\cup, \cap)$ and the arithmetic inequality $(\geq)$.

Constraints	Interpretation
r `bin_r` d `-->` a	$r =$ `birel`(l, d, a) where $l = \{s_i \mid \ \forall i \in d, \ s_i \ \in \{\}..a\}$
(i, j) `in_r` r	if $i \in d, j \in a$ then $j \in s_i$
(i, j) `notin_r` r	if $i \in d, j \in a$ then $j \notin s_i$
`funct`(r)	$\forall i \in d, \ \#s_i = 1$
`inj`(r)	$\#d \leq \#a, \ \#d = n$ $s_1 \cap s_2 = \emptyset, \ s_1 \cap s_3 = \emptyset, \ ..., \ s_{n-1} \cap s_n = \emptyset$ $\forall i \in d, \ \#s_i = 1$
`surj`(r)	$\#d \geq \#a, \ \#d = n$ $s_1 \cup s_2 ... \cup s_n = a$ $\forall i \in d, \ \#s_i = 1$
`bij`(r)	$\#d = n, \ \#a = n$ $s_1 \cap s_2 = \emptyset, \ s_1 \cap s_3 = \emptyset, \ ..., \ s_{n-1} \cap s_n = \emptyset$ $\forall i \in d, \ \#s_i = 1$

These constraints do not require any specific solver since the reasoning is based on the successor set variables.

Example 82 *The goal:*

```
:- R bin_r {1, 2} --> {a, b, c}, funct(R).
```

creates the term:

```
R = birel([Set1{[{},{a,b,c}]}, Set2{[{},{a,b,c}]}], {1,2}, {a,b,c})
```

and the list of delayed goals:

```
#(Set1{[{},{a,b,c}]}, 1), #(Set2{[{},{a,b,c}]}, 1)
```

Since the created compound term is not visible to the user, a collection of predicate relations allow him to access to the properties of the relation:

- `succs`(r, l) instantiates l to the list of successor sets of r.
- `dom`(r, s) instantiates s to the DS-domain of r.
- `ran`(r, s) instantiates s to the AS-range of r.
- `succ`(r, e, s) instantiates s to the successor set of the element e belonging to DS-domain, such that $s = \{x \mid (e, x) \in r\}$.

5.2 Implementation of Conjunto

The implementation of Conjunto was done in the ECL^{i}PS^e [ECR94] system which extends the plain Prolog language with features dedicated to the implementation of specific constraint solvers. The main features provided at the language level comprise the attributed variable data structure and the suspension handling predicates. An attributed variable is a special data type [Hui90][Hol92] which consists of a variable with a set of attributes attached and whose behaviour on unification can be explicitly defined by the user in a way that differs from Prolog unification. Attributed variables aim at dealing with specific computation domains distinct from the Herbrand universe. The suspension handling predicates provide means to (1) delay a goal or constraint, (2) store it in a specific list with respect to one or several variables, (3) awake a list of delayed goals when some given conditions are satisfied. The suspension handling predicates allowed us to implement the data driven constraint handling in Conjunto. In addition, the Conjunto solver makes use of the finite domain library of ECL^{i}PS^e to deal with integer interval terms (as well implemented as attributed variables).

5.2.1 Set data structure

A set variable is not represented as a standard Prolog variable, but as an attributed variable which is subject to a dedicated unification algorithm. The internal representation of ground sets is also given since it influences the time complexity of the transformation rules. Both the data structure and the internal representation of ground sets are not visible to the user and will be ignored in the description of the transformation rules.

5.2.1.1 Set variable representation

A set variable is an attributed variable comprising the following list of attributes. This structure stores for each set variable all the necessary information regarding its domain, cardinality, and weight (null if undefined) together with three suspension lists. The attribute arguments have the following meaning:

- **setdom: [Glb,Lub]** represents the set domain. The user can access it using the built-in predicates `glb`, `lub`.
- **card: C** represents the set cardinality. This attribute **C** is initialized as soon as a set domain is attached to a variable. It is either an integer interval or an integer. It can be accessed and modified using specific built-in predicates from a finite domain library.
- **weight: W** represents the set weight. **W** is intialized to zero if the domain is not a weighted set domain, otherwise it is computed as soon as a weighted set domain is attached to a set variable. It can be accessed and modified using specific built-in predicates from a finite domain library.
- **del_glb: Dglb** is a suspension list that should be woken when the lower bound of the set domain is updated.
- **del_lub: Dlub** is a suspension list that should be woken when the upper bound of the set domain is updated.
- **del_any: Dany** is a suspension list that should be woken when any set domain refinement is performed.

5.2.1.2 Ground set representation

The choice for the internal representation of sets is independent of the algorithms, and not visible to the user. However, it plays a role in the time complexity of the different set operations. In contrast to integer intervals, the time complexity for operations on ground sets ($+$, $-$ versus $\cup$, $\cap$, $\backslash$) can not be considered as constant for it closely depends on the internal representation of a set. In Conjunto each ground set is represented by a sorted list where the time complexity for any set operation ($\cup$, $\cap$, $\backslash$) is bounded from above by $\mathcal{O}(2d)$ where d is $\#lub(s) + \#glb(s)$ and s the set with the largest domain.

Since we work essentially on set domains, another approach has been tried out which consists in representing a set domain as a boolean vector mapped onto a list containing the actual value of the elements. The upper bound is specified by the set of elements whose corresponding 0-1 variable has the value 1 or 0-1 (undetermined). The lower bound is specified by the set of elements whose corresponding 0-1 variable has the value 1. This approach reduces the time complexity of the $\cup$ and $\cap$ operations to $\mathcal{O}(\#lub(s))$ where $lub(s)$ is the largest domain upper bound. But this leads to much larger memory usage due to the size of the domains used in practice and to the handling of two lists (the list of 0-1 variables and the list of actual values).

From now on, the value of d in the complexity results will always stand for $\#lub(s) + \#glb(s)$.

5.2.2 Set unification procedure

A Conjunto program attaches a specific semantics to set terms. This semantics requires to extend the Prolog unification to the one of set terms. The behaviour of the set unification procedure comprises the following tests and inferences:

- the unification of a logical variable and a set variable. The logical variable is bound to the set variable.
- the unification of a ground set and a set variable. The set variable is instantiated to the ground set if it belongs to its domain.
- the unification of two set variables. The two variables are bound to a new variable whose domain is the convex intersection of the two domains (cf. set interval calculus). If this domain is empty the unification fails.
- the unification of a set variable with any other term fails.

The unification procedure is used in the generic algorithm for a system of Conjunto constraints. It will be implicitly referred to by the connective $\leftarrow$.

5.2.3 Local transformation rules

Consistency notions for primitive set constraints and graduated constraints have been defined in the formal part (cf. 3.3.3). By making use of these definitions, the following transformation rules check and infer the consistency of primitive Conjunto constraints. They are based on interval reasoning techniques which are approximations of the constraint satisfaction techniques. The basic idea consists in pruning the set domains attached to the set variables by removing set values which can never be part of any feasible solution. Set values are removed by adding elements to the lower bound of the domain and/or by removing elements from the upper bound.

5.2.3.1 Transformation rules for primitive set constraints

Primitive set constraints are s `` `< `` s_1 and s `` `<> `` s_1 where s and s_1 denote set variables ranging over a set domain. The transformation rules are depicted in figure 5.3.

Consider the set inclusion constraint s_1 `` `< `` s_2 such that $s_1 \in d_1$, $s_2 \in d_2$. The transformation rule makes use of the lower and upper ordering of the set inclusion. Making this constraint consistent might require adding elements to the lower bound of the domain d_2 and removing elements from the upper bound of d_1. The refinements lead to the new domain bounds:

T1.	$\mathtt{glb}(d'_1) \leftarrow \mathtt{glb}(d_1)$	$\mathtt{lub}(d'_1) \leftarrow \mathtt{lub}(d_1) \cap \mathtt{lub}(d_2)$
T2.	$\mathtt{glb}(d'_2) \leftarrow \mathtt{glb}(d_2) \cup \mathtt{glb}(d_1)$	$\mathtt{lub}(d'_2) \leftarrow \mathtt{lub}(d_2)$

Consider the disjointness constraint s_1 `` `<> `` s_2 such that $s_1 \in d_1$, $s_2 \in d_2$. The only possible refinement aims at removing elements from each upper bound of a set domain which are definite elements that belong to the other set. This constraint is consistent if the refined domains for the variables are:

T3.	$\mathtt{glb}(d'_1) \leftarrow \mathtt{glb}(d_1)$	$\mathtt{lub}(d'_1) \leftarrow \mathtt{lub}(d_1) \setminus \mathtt{glb}(d_2)$
T4.	$\mathtt{glb}(d'_2) \leftarrow \mathtt{glb}(d_2)$	$\mathtt{lub}(d'_2) \leftarrow \mathtt{lub}(d_2) \setminus \mathtt{glb}(d_1)$

Figure 5.3 Interval refinement for primitive set constraints

Thanks to the monotony of the set operations $(\cup, \cap)$, the interval reasoning applied is equivalent to domain reasoning *i.e.*, it guarantees that each element in the domains is a possible value for the set.

Complexity issues. The time complexity for each transformation is bounded by $\mathcal{O}(d)$ since only one set operation is applied each time.

5.2.3.2 Projection functions for n-ary constraints

Constraints over set expressions have not been dealt with so far. These n-ary constraints require a special handling mechanism due to the properties of the set operations. *If there is more than one set operation in the constraint, it is practically impossible to express each set variable in terms of the others, since set operations have no direct inverse.* This point requires us to tackle n-ary constraints as "mini-programs". The approach implemented in Conjunto consists in approximating an n-ary constraint by (1) associating each basic set expression (s_1 `\/` s_2, s_1`/\` s_2, s_1`\`s_2) with its relational form, (2) applying inductively this process until the n-ary constraint can be expressed as a binary one. The relational forms of set expressions are derived by creating a new set variable whose domain is approximated by using the set interval calculus. The relational forms correspond to the following constraints:

$$\begin{array}{lcl} \texttt{union}\ (s_1, s_2, s) & \leftrightarrow & s_1 \ \texttt{\textbackslash/}\ s_2\ \texttt{`=}\ s \\ \texttt{inter}\ (s_1, s_2, s) & \leftrightarrow & s_1 \ \texttt{/\textbackslash}\ s_2\ \texttt{`=}\ s \\ \texttt{diff}\ (s_1, s_2, s) & \leftrightarrow & s_1 \ \texttt{\textbackslash}\ s_2\ \texttt{`=}\ s \end{array}$$

The local consistency of these 3-ary constraints ensures that no triples satisfying the constraint are excluded. The inference is performed using transformation rules that make use of the projection functions each of whose describing each set domain in terms of the others (cf. formal part 3.3.3). Each such projection uniquely defines a smallest set domain which contains the possible solution values. Three projection functions are required per relational constraint. They are depicted in figures 5.4, 5.5, 5.6.

Projection functions associated to the constraint `union`(s_1, s_2, s) *such that* $s_1 \in d_1, s_2 \in d_2, s \in d$. *T5 holds also for* s_2.

$$\begin{array}{lll} \text{T5.} & \texttt{glb}(d'_1) & \leftarrow \texttt{glb}(d_1) \cup \texttt{glb}(d) \setminus \texttt{lub}(d_2) \\ & \texttt{lub}(d'_1) & \leftarrow \texttt{lub}(d_1) \cap \texttt{lub}(d) \\ \text{T6.} & \texttt{glb}(d') & \leftarrow \texttt{glb}(d) \cup \texttt{glb}(d_1) \cup \texttt{glb}(d_2) \\ & \texttt{lub}(d') & \leftarrow \texttt{lub}(d) \cap \texttt{lub}(d_1) \cup \texttt{lub}(d_2) \end{array}$$

Figure 5.4 Projection functions associated to the set union relation

The union of two sets represents a logical disjunction. So it is very unlikely that the addition of new elements to $\texttt{glb}(d)$ requires modifying the lower bound of the domains of s_1 or s_2. The one case which requires such a refinement occurs if some elements belong to the lower bound of d and can never belong to one of the two sets (cf. T5). Consequently they should be added to the other one.

Projection functions associated to the constraint `inter`(s_1, s_2, s) *such that* $s_1 \in d_1, s_2 \in d_2, s \in d$. T7. *holds also for* s_2.

$$\begin{array}{lll} \text{T7.} & \texttt{glb}(d'_1) & \leftarrow \texttt{glb}(d_1) \cup \texttt{glb}(d) \\ & \texttt{lub}(d'_1) & \leftarrow \texttt{lub}(d_1) \setminus ((\texttt{lub}(d_1) \cap \texttt{glb}(d_2)) \setminus \texttt{lub}(d)) \\ \text{T8.} & \texttt{glb}(d') & \leftarrow \texttt{glb}(d) \cup \texttt{glb}(d_1) \cap \texttt{glb}(d_2) \\ & \texttt{lub}(d') & \leftarrow \texttt{lub}(d) \cap \texttt{lub}(d_1) \cap \texttt{lub}(d_2) \end{array}$$

Figure 5.5 Projection functions associated to the set intersection relation

The intersection of two sets represents a logical conjunction. So any addition of elements to one of the three domains requires modifying at least one of the lower bounds of the domains. A pruning of the upper bound of these domains is rarer. However, it might occur in the case depicted in T7 which corresponds to the following configuration: some elements are definite ones of s_2 (or s_1) and possible ones of s_1 (or s_2). If they cannot belong to s then they should be removed from the upper bound of the domain of s_1 (respectively s_2).

Projection functions associated to the constraint $\mathtt{diff}(s_1, s_2, s)$ *such that* $s_1 \in d_1, s_2 \in d_2, s \in d$:

T9.	$\mathtt{glb}(d_1')$	$\leftarrow$	$\mathtt{glb}(d_1) \cup \mathtt{glb}(d)$
	$\mathtt{lub}(d_1')$	$\leftarrow$	$\mathtt{lub}(d_1) \setminus (\mathtt{lub}(d_1) \setminus (\mathtt{lub}(d) \cup \mathtt{lub}(d_2)))$
T10.	$\mathtt{glb}(d_2')$	$\leftarrow$	$\mathtt{glb}(d_2)$
	$\mathtt{lub}(d_2')$	$\leftarrow$	$\mathtt{lub}(d_2) \setminus \mathtt{glb}(d)$
T11.	$\mathtt{glb}(d')$	$\leftarrow$	$\mathtt{glb}(d) \cup \mathtt{glb}(d_1) \setminus \mathtt{glb}(d_2)$
	$\mathtt{lub}(d')$	$\leftarrow$	$\mathtt{lub}(d) \cap \mathtt{lub}(d_1) \setminus \mathtt{glb}(d_2)$

Figure 5.6 Projection functions associated to the set difference relation

The second part of the rule T9 considers a particular case where the upper bound of d_1 should be pruned. If $\mathtt{lub}(d_1)$ contains elements which do not belong both to the upper bound of d *and* to the upper bound of d_2, then these elements cannot belong to s_1. Both conditions must be satisfied to prune $\mathtt{lub}(d_1)$.

Complexity issues. Time complexity for each transformation rule is bounded by $\mathcal{O}(d)$ times the number of basic set operations, which is bounded by 4 for the rules T7 and T9.

Remark. The relational constraints are transparent to the user at the programming level. However, any temporary state of a program is given in terms of these newly created constraints.

Example 83 *A partially solved constraint of the form:* `S1 \/ S2 ‘< S2 /\ S3` *is stored using the set of delayed goals:*

```
union(S1, S2, S12),
inter(S2, S3, S23),
S12 ‘< S23.
```

5.2.3.3 Graduated constraints: cardinality and weight constraints

Graduated constraints deal with set variables and integer variables. Inferring the partial consistency of these constraints might require refining the integer domains or assigning a value to a set. Since graduations are not bijective functions, a modification of the integer domains is not a sufficient condition to require a set domain refinement. The pruning achieved by the following transformation rules guarantees that (1) the values removed from the domains cannot be part of any feasible solution, (2) if a solution exists, its value lies in the remaining set and integer domains.

Consider the set cardinality constraint $\#(s, x)$ *where* $s \in d$ *and* $x \in [m, n]$*.* x *is an integer variable. We have:*

T12.	$[m', n']$	$\leftarrow [m, n] \cap [\#\texttt{glb}(d), \#\texttt{lub}(d)]$
T13.	d'	$\leftarrow \texttt{glb}(d)$ if $\#\texttt{glb}(d) = n$
T14.	d'	$\leftarrow \texttt{lub}(d)$ if $\#\texttt{lub}(d) = m$

Figure 5.7 Transformation rules for the set cardinality constraint

The transformation rules for the weight constraint are similar. The only difference lies in the initial computation of the integer intervals.

Consider the weight constraint $\texttt{weight}(s, y)$ *where* $s \in d, y \in [m, n]$ *and* $\sum_{(e_k, m_k) \in \; glb(d)} m_k = w_{glb}$ *and* $\sum_{(e_k, m_k) \in \; lub(d)} m_k = w_{lub}$*. We have:*

T12'.	$[m', n']$	$\leftarrow [m, n] \cap [w_{glb}, w_{lub}]$
T13'.	d'	$\leftarrow \texttt{glb}(d)$ if $m = w_{lub}$
T14'.	d'	$\leftarrow \texttt{lub}(d)$ if $n = w_{glb}$

Figure 5.8 Transformation rules for the weight constraint

5.2.4 Constraint solver

The transformation rules described so far deal with individual constraints. The constraint solver applies these rules to check/infer the consistency of an admissible system of constraints in an incremental way. Incrementality refers to the nature of the Conjunto solver which stores each newly consistent constraint and handles the consistency of each constraint in conjunction with the constraint store.

The algorithm. Let a tuple $(c, \vec{s})$ denote a constraint c over a set of variables designated by $\vec{s}$. The initial set of constraints to be considered is designated by G. A list C which represents the constraint store contains all the constraints whose consistency has been checked. The solver selects one constraint c at a time in G and applies to it the adequate local transformation rule using a depth first search strategy. Each constraint c is determined to be consistent if the transformation rule infers consistent domains. This might require some domain refinements and consequently a need to reconsider some constraints in C whose variables intersect with those in c. Such constraints are moved from C to G. This process describes the data driven mechanism of the solver. The constraint c is then added to the constraint store C and another constraint is selected in G. The last state of the resolution is reached once no goal remains in G, or when a failure is encountered (*i.e.*, at least one set domain $[a, b]$ or integer interval $[m, n]$ is such that $a \not\subseteq b$ or $m \not\leq n$). The program returns the set of constraints C which are locally consistent. The general schema of the algorithm is depicted in figure 5.9.

begin
    Initialize G to the list of all the constraints in the admissible system
    Initialize C to the empty list
    while G is not empty **do**
        begin
            select and remove the first constraint $(c, \vec{s})$ from G
            apply the adequate transformation rule on $(c, \vec{s})$ which returns $(c, \vec{s'})$
            if $\vec{s'}$ is inconsistent **then**
              exit with failure
              else if $\vec{s} \neq \vec{s'}$ **then**
                begin
                    $\vec{s} \leftarrow \vec{s'}$
                    for each $(p, \vec{v})$ in C **do**
                        if $\vec{s'} \cap \vec{v} \neq \emptyset$ **then**
                            remove $(p, \vec{v})$ from C and add it to G
                end
              add $(c, \vec{s})$ to the end of C.
        end
end

Figure 5.9 General algorithm

This generic algorithm generalizes the complete algorithm we have described in [Ger94] by moving from the handling of a system containing only primitive set constraints to a system containing any constraint allowed in the language. This algorithm resembles the relaxation algorithm used by CLP(Intervals) systems [LvE93] also referred to as fixed point algorithm in [BMH94] [Ben95]. All of those can be seen as an adaptation of the AC-3 algorithm [Mac77] where domains are specified by intervals. The only difference between the algorithms lies in the transformation rules applied. The generic algorithm satisfies the following properties of fixed point algorithms.

Theorem 84 *The algorithm always terminates.*

Proof (termination) This comes from the fact that the domains are finite and only get refined: in the different transformation rules, the new lower bounds are computed by extending the former ones (union operation) and the upper bounds are derived by intersecting or removing elements from the former ones. If an inconsistency is detected, the algorithm terminates with failure.□

Theorem 85 *If a solution exists, it can be derived from the simplified system of constraints.*

Proof This follows directly from the monotony of the convex closure operators[6] and the inferences performed in the transformation rules. Monotony guarantees that the actual value of a set or integer lies in the approximated domains. The transformation rules aim at removing values which can never be part of any feasible solution. So all possible solution values are kept.□

Complexity issues Let l be the size of G and e the size of C. The cost of one transformation rule is bounded by $\mathcal{O}(6d)$ (d being the largest $\#lub(s) + \#glb(s)$). For one constraint the algorithm can be iterated at worst d' times if $d' = \#lub(s) - \#glb(s)$. If these iterations are necessary for all the constraints the worst time complexity is then $\mathcal{O}(ldd')$.□

This time complexity does not occur in practice. On the one hand, if it occurs this means the algorithm leads to a complete solution which is quite rare. On the other hand, the constraints are not systematically reconsidered if some of their variable domains get modified. Indeed, the constraints are stored in various suspension lists so as to avoid reconsidering them when there is no need to do so. These lists are described below.

[6] They have been described in the formal part 3.2.3

5.2.4.1 Suspension lists

Three different lists are attached to each set variable. They are meant to improve the time complexity and thus the efficiency of the solver by splitting the list C so that only those constraints concerned with the specific domain refinement are woken. Corresponding to each set variable s_i with domain d_i, each of the three lists could contain the following goals:

- Q_{glb} contains the primitive constraints for which a modification of the lower bound of d_i might require reconsidering the constraints. It contains only constraints of the form s_i `` `< `` s_j.
- Q_{lub} contains the primitive constraints for which a modification of the upper bound of d_i might require reconsidering the constraints. It contains the constraints of the form: s_j `` `< `` s_i, s_i `` `<> `` s_j, (and its symmetrical s_j `` `<> `` s_i).
- Q_{any} contains the remaining constraints for which any set domain modification might require reconsidering them. In other words it contains the relational constraints (relational forms of the set union, intersection and difference operations) and the graduated constraints in which the variable s_i appears.

In addition, the graduated constraints are also stored in the list of delayed goals attached to the integer variables appearing in it. While graduated constraints are delayed only once, they are attached to two lists and thus might be reactivated with respect to two different conditions. This process establishes the dynamic cooperation between the Conjunto solver and the finite domain solver. It guarantees that the partial consistency of a graduated constraint is always maintained within a constraint system.

5.2.5 Execution of a Conjunto program: architecture

The Conjunto solver can be embedded in any logic-based language provided a set of constraint solving facilities is given or can be defined. These facilities comprise (1) attributed variables or a similar structure which links a set variable to its domain and the required lists of delayed goals, (2) suspension handling mechanisms to deal with delayed goals, (3) possibly a finite domain library to tackle set based optimization problems. Figure 5.10 presents the execution of a Conjunto program together with the different modules and functionalities required.

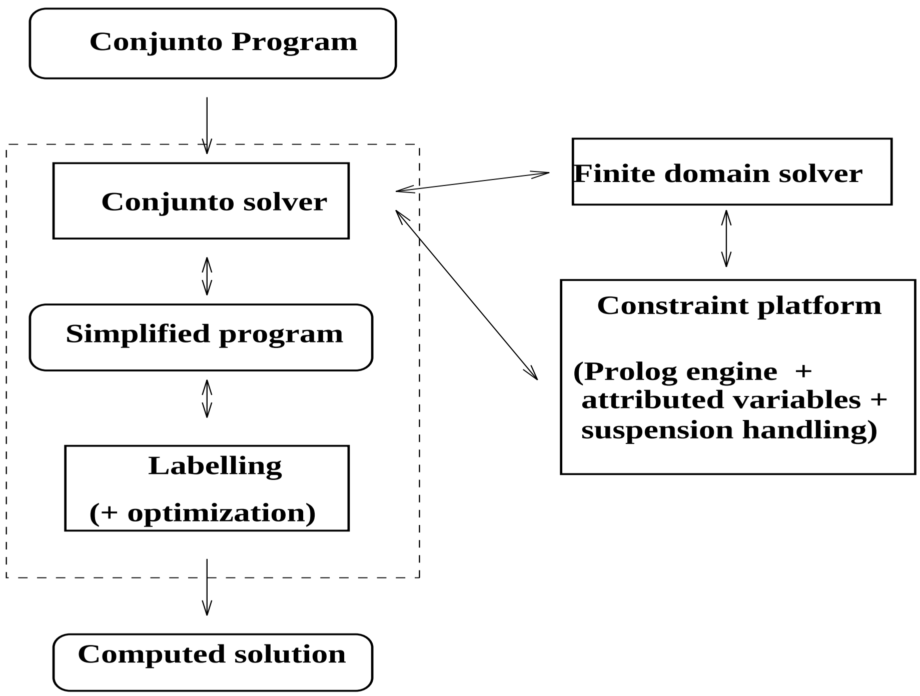

Figure 5.10 Execution of a Conjunto program

6
Applications

Tout ce que vous faites maintenant
est acte de rêve, pensée de rêve.
Que vos rêves soient toujours de plus en plus beaux !
Car tout deviendra réalité.

This chapter shows the applicability of the Conjunto language to the modelling and solving of set based search problems. We describe how combinatorial search problems can be modelled as set domain constraint satisfaction problems using the Conjunto language. The focus is on the expressiveness and the efficiency of the language when dealing with search problems and optimization problems arising from operations research and combinatorial mathematics.

6.1 Set domain CSPs

The modelling and solving of a set domain CSP follows the usual procedure for CSPs which consists of the problem statement, the labelling procedure and possibly the search for an optimal solution.

6.1.1 Problem statement

The statement of a set domain CSP amounts to:

- Initializing the set variables by assigning a set domain to them.
- Stating the constraints. The constraints can be set constraints or graduated constraints. The set constraints establish links between set variables. The graduated constraints restrict the possible set of values a set could take by applying a kind of measure to the set. The set cardinality constraint is used to bound the cardinality of a set to a specific integer domain (or

possibly to an integer). The weight constraint restricts the sum of the integer values appearing in a set domain. These constraints might generate integer variables which are not relevant for the final solution, but which take part in the problem definition and particularly in optimization functions.

6.1.2 Labelling

The labelling phase aims at finding values for the distinguished set variables [MR93], that is those which are part of the final solution. This can be done either by using the pre-defined labelling procedure `refine` described in the practical framework (cf. 4.1.4.3), or by defining a new labelling procedure based on specific labelling strategies. An efficient set labelling procedure should not try to directly instantiate a set to one of its domain elements. The reason is that by doing so, the satisfaction of those constraints for which only a partial consistency is guaranteed is reached in a passive way. The best method in terms of active use of (graduated) constraints is based on incremental set domain refinements by adding one by one elements to the lower bound of the set domain (or possibly by removing elements from the upper bound)

6.1.3 Optimization

The concept of optimality is related to the notion of minimizing or maximizing a cost function. This function necessarily denotes a measure, takes as input an arithmetic expression and returns an integer value. Possible cost functions associated with a set domain CSP are the sum of the set cardinality values, the sum of the weights, etc. Such a function constrains the sets via their associated measure and consequently no specific optimization predicate is required to deal with sets. The user can make use of existing predicates developed for integer domain CSPs with an optimization criterion. One of these predicates used in a subsequent application (set partitioning), performs the branch and bound search.

The predicate `min_max(Goal, Cost)` searches for a solution to the goal `Goal` that minimizes the value of the linear term `Cost` using the branch and bound method from operations research [PS82]. As soon as a partial solution to `Goal` is found whose cost is worse than the previous solution the search is not explored any further and a new solution is searched for.

Another predicate is often used to minimize the cost of a solution within a fixed range: `min_max(Goal, Cost, Min, Max, Percent)`. This predicate also makes use of the branch and bound method with some restrictions. It starts with the assumption that the value `Cost` to be minimized is less than or equal to

`Max`. As soon as a solution is found whose minimized value is less than `Min`, this solution is returned. When one partial solution is found, the search for the next better solution starts with a minimized value `Percent` % less than the previous one.

The use of these predicates in a set domain CSP requires the definition of `Goal` as a set labelling procedure call, plus a graduated constraint whose integer value is `Cost`. The solving of `min_max/2/5` will execute the labelling procedure and incrementally refine the integer domain involved in the graduated constraint. Once all the sets are labelled the integer domain becomes one value (the cost) which can be evaluated. The optimization process will then constrain the integer variable appearing in the graduated constraint to have its value in a new domain whose upper bound is lower than the cost previously computed.

6.2 Modelling facilities

The two problems presented in this section come from the areas of combinatorial mathematics [Lue89] and operations research. The first one —the ternary Steiner problem— is to find a specific hypergraph whose nodes are integer variables. Our approach illustrates how an hypergraph whose nodes are integer variables can be modelled as a simple graph whose nodes are set variables. The second problem is a set partitioning problem usually represented by mathematical models and solved using integer linear programming techniques. Here it is modelled as a set domain CSP.

6.2.1 Ternary Steiner problem

The ternary Steiner problem has its origins in combinatorial mathematics. It belongs to the class of block theory problems which deal with the computation of hypergraphs. A hypergraph is a graph with the property that some arcs connect collections of nodes. This problem has only recently been addressed in computer science. [Bel90b] addresses this problem for the first time. The approach consists in representing the problem as an integer domain CSP in a constraint logic programming (CHIP [DSea88]), using the new concept of global constraints. The integer domain CSP modelling corresponds to the hypergraph representation: the integer variables represent the nodes and the global constraints represent the hyperarcs.

Problem statement The statement is taken from [Bel90b]. A ternary Steiner system of order n is a set of $T = n(n-1)/6$ triples of distinct elements in $\{1, ..., n\}$ such that any two triples have at most one element in common. The mathematical properties of this problem prove that n modulo 6 has to be equal to 1 or 3 [LR80]. One solution of Steiner 7 is for example:

$$\{1,2,6\}, \{1,3,5\}, \{2,3,4\}, \{3,6,7\}, \{2,5,7\}, \{1,4,7\}, \{4,5,6\}$$

The integer domain CSP modelling or hypergraph representation uses three nodes, or variables, ranging over $\{1, ..., n\}$ to represent a triple $\{X, Y, Z\}$. The constraints are (1) ordering constraints between the three nodes ($X < Y < Z$) so as to remove equivalent triples under permutations of the elements; (2), any triple must have at most one element in common with the other triples of nodes. This amounts to constraining each pair of a triple to be pairwise distinct from any other pair appearing in another triple. This requires constraining all the $n(n-1)$ possible pairs (6 per triple [X, Y, Z]: [X,Y], [Y,X], [X,Z], [Z,X], [Y,Z], [Z,Y]) to be pairwise distinct. This approach is sound but far too costly in variables and constraints. A global constraint `all_pair_diff` has been defined in [Bel90a][Bel90b] to free the user from specifying all the pairwise distinct pairs.

If each set of three nodes, describing a triple, can be represented as one variable, then the hypergraph corresponds to a graph. This allows the modelling to be simpler and to require less variables. Such a modelling corresponds to a set domain CSP approach.

Problem modelling Modelling the problem as a set domain CSP involves representing each triple as *one* set variable. Let $S_i, 1 < i < T$ denote the T set variables which represent the triples. Their domains are initialized to the set domain `[{},{1,...,n}]`.

The constraint "any two triples have at most one element in common" is simply represented by: `#(` S_i `/\` S_j `) =< 1`. The constraint generation is summed up in the short program:

```
constraints(Lsets) :-
     card_all(Lsets, 3),
     intersect_atmost1(Lsets).

card_all([], N).
card_all([Set1|LSets], N) :-
     #(Set1, N),
     card_all(LSets, N).

intersect_atmost1([]).
intersect_atmost1([S1 |L]) :-
     distinctsfrom(S1, L),
     intersect_atmost1(L).

distinctsfrom(_S, []).
distinctsfrom(S, [S1 | L]) :-
     #(S /\ S1, C), C =< 1,
     distinctsfrom(S, L).
```

`card_all` constrains the cardinality of each set variable in the list `Lsets` to be equal to 3. The predicate `intersect_atmost1` generates the main constraint to be satisfied by each pair of triples.

Problem solving The resolution makes use of the labelling procedure `refine(S)` for each triple `S`. If $n = 7$, the first set is instantiated to $\{1, 2, 3\}$. Then the system tries to instantiate the second set by first adding the element 1 to its lower bound. This domain refinement requires reconsidering the constraint `#(S1 /\ S2, C), C =< 1`. This results in a refinement of the domain of `S2` by a removal of the values 2 and 3 from the upper bound of its domain. At this stage in the resolution, the refined domains are:

```
S1 = {1,2,3}, S2 `:: [{1},{1,4,5,6,7}],
[S3,S4,S5,S6,S7] `:: [{},{1,...,7}].
```

Computation results The problem was solved in 0.8 sec on a Sun4/40 for $n = 7$. Six choice points were created during the solution step. Beldiceanu [Bel90b] says that 21 choice points were generated and 0.08 sec were sufficient to solve the problem. This difference in choice points and time was surprising. Unfortunately the global constraint and the program developed were not available and so, in order to make a sound comparison, we developed the same program as described in the paper using the ECL^{i}PS^e integer domain library. The choice points and the time required were then similar to the Conjunto approach, but the program was much less natural.

The complexity of this problem grows exponentially with n. In [Bel90b] the problem has not been tackled for larger values than 7. Indeed, it turned out that using the same program to solve the problem when $n = 9$ leads to a combinatorial explosion. We defined a labelling strategy which consists in constraining each element to belong to at most $(n - 1)/2$ triples. Indeed, there are at most $n - 1$ distinct pairs containing one element i and a triple containing i must contain 2 of these pairs. In practice this labelling strategy corresponds to a simple occur check before adding one element to a set domain. This does not help when $n = 7$ but for $n = 9$ it reduced the number of choice points from 7180 to 116 and consequently the computation time from 501 sec. to 18 sec.

Remark. For one value of n there exists more than one solution. The search for all the possible solutions requires us to take into account the symmetries inherent to the problem *i.e.*, those which do not depend on the modelling. A permutation of two sets does not change the actual solution but corresponds, from a computational point of view, to new instances of the set variables. In fact, the modelling of a search problem as a set domain CSP removes the symmetries that come from an integer domain CSP approach. In the Steiner application, the solving of the set domain CSP program led to a pruning of the search space which is equivalent to that achieved by the global constraints, aiming at removing local symmetries. Consequently, set constraints resemble some global constraints in terms of problem solving and pruning ability, but to cope with this actual symmetries of the problem a global reasoning on sets is necessary.

6.2.2 The set partitioning problem

The set partitioning problem [GM84] is an optimization problem that comes from operations research. Consider a mapping from a set of elements to a collection of equivalence classes each of which contains a subset of these elements, and has a specific cost. The objective is to find a subset of the classes such that they are all pairwise disjoint, each element is mapped onto exactly one class and the total cost of the selected classes is minimal. The set partitioning problem resembles the set covering problem, but it is more complex because the disjointness constraints do not guarantee that a feasible solution exists.

This problem is currently tackled as a 0-1 integer linear programming problem using the following mathematical model:

$$\text{minimize } (c * x), \quad (a_{ij})\ x = e_m$$

where c is a cost vector $1 * n$, (a_{ij}) is an $m * n$ known matrix comprising 0 and 1 values, x is an $n * 1$ vector of 0-1 variables and e_m is a vector of m entries equal to 1. We have:

$$\forall i \in Dom, \forall j \in \{1, ..., n\},\ a_{ij} = \begin{cases} 1 & \text{if } i \in S_j, \\ 0 & \text{otherwise} \end{cases}$$

Each equivalence class is denoted by a set S_j.

Example 86 *A 0-1 modelling corresponds to the following system of constraints:* *min* $c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 + c_5x_5 + c_6x_6$

$$\begin{array}{cccccccc} x_1+ & & x_3+ & & x_5 & & = 1 \\ x_1+ & x_2+ & x_3+ & & & & = 1 \\ x_1+ & & x_3+ & & & x_6 & = 1 \\ & & & x_4+ & x_5+ & x_6 & = 1 \\ & & & x_4+ & & x_6 & = 1 \end{array}$$

Each column represents an equivalence class. Each line refers to one element in $\{1,..,5\}$*. The equality constraints specify that an element can belong to exactly one equivalence class.*

Problem statement The mathematical statement of the problem is depicted here in terms of relations and set constraints. Consider a mapping R from Dom to Ran which is constrained to be an application. Let the DS-domain be $Dom = \{1, 2, ..., m\}$ and the AS-range be a family Ran of n subsets of Dom such that $Ran = \{S_1, ..., S_n\}$ where each S_j is an equivalence class (a ground set) and:

$$\bigcup_{j \in \{1,2,...,n\}} S_j = Dom$$

A subset P_0 of Ran is a partition of Dom if and only if:

$$\bigcup_{j \in \{1,2,...,n\}} S_j = Dom \bigwedge \forall S_j, S_k \in P_0, S_j \cap S_k = \emptyset$$

A cost set S_c is associated to the elements S_i of Ran by considering a weighted set composed of elements (S_i, w_i). The final problem is to determine a partition P^* such that:

$$\sum_i w_i \text{ is minimal}$$

This statement corresponds to the approach used with the Conjunto language.

Problem modelling Let a relation R on the ground sets Dom and Ran be constrained to be an applicative mapping. Each successor set is constrained to be a subset of the proposed sets. These constraints are not sufficient to solve the problem. Two other requirements are necessary:

- the final set P^* of equivalence classes should contain only disjoint sets.
- an instantiated successor set should also represent the successor set of all its predecessors.

This corresponds to adding two constraints which will be checked using the forward checking inference rule (*i.e.*, once a successor set becomes ground). Informally, as soon as one successor set $\texttt{succ}(R, i, \{s_k\})$ becomes ground we must have:

$$\forall j \in Dom, \texttt{succ}(R, j, s_j) \begin{cases} \text{if } j \in s_k, & s_j = \{s_k\} \\ \text{if } j \notin s_k, & s_j \cap \{s_k\} = \emptyset \end{cases} \quad (1)$$

These constraints correspond to the program:

```
disj_or_eq(_R, _Dom, []).
disj_or_eq(R, Dom, [S | LSuccs]) :-
        (set(S), S = {Eq}
            ->
          iterate(Eq, E, (succ(R,E, {Eq}))),
          Diffset `= Dom \ Eq,
          succ(R,F,Sf),
          iterate(Diffset, F, (Eq notin Sf))
        ;
          delay(disj_or_eq(R, [S]), S, glb)),
/* the constraint is delayed and woken when the lower bound of S
   gets modified */
        disj_or_eq(R, Dom, LSuccs).
```

`disj_or_eq` generates the constraints (1) which should be satisfied by each successor set. It takes as input the application `R`, its domain `Dom` and the list of all the successor sets `[S | LSuccs]`. The constraint `disj_or_eq(R, [S])` is delayed if the successor set `S` is not ground, and activated as soon as it becomes ground. The `iterate(S, E, Goal)` predicate is an abbreviation for purposes of clarity only. Its role is to apply to each element `E` in the ground set `S` the goal `Goal`. At the implementation level, it transforms the ground set S into a list and iterates over this list.

Example 87 *The statement of the above example using Conjunto corresponds to the following set of constraints:*

```
R bin_r {1,2,3,4,5} --> {{1,2,3},{2},{1,2,3}, {4,5},{1,4},{3,4,5}},
appl(R),
succ(R, 1, S1), S1 `< {{1,2,3},{1,4}},
succ(R, 2, S2), S2 `< {{1,2,3},{2}},
succ(R, 3, S3), S3 `< {{1,2,3},{3,4,5}},
succ(R, 4, S4), S4 `< {{4,5},{3,4,5}},
succ(R, 5, S5), S5 `< {{4,5},{3,4,5}},
/* each element i is mapped to a set Si whose domain contains the
   possible equivalence classes (ie. those which contain i) */
/* Note that columns 1 and 3 in the ILP modelling correspond here
   to one equivalence class {1,2,3}*/
disj_or_eq(R, {1,2,3,4,5}, [S1,S2,S3,S4,S5]).
```

The search space associated to these problems is usually very large and simplification rules are applied in order to reduce the initial problem size. An overview of these rules can be found in [HP92] [Pad79]. They consist in removing rows and columns in the adjacency matrix formulation. This corresponds to removing, in a deterministic manner, redundant sets from the successor set domains, and to bounding some successor sets to the same variable. The main operations amount to checking disjointness and/or inclusion of sets and to computing cliques over the successor set domains.

The set of rules corresponds to the following sequences of computations: (1) compute the clique K_i in the associated intersection graph of R attached to each element i in Dom. This means: for each successor set S_i attached to i, collect all the sets in Ran which have at least one element in common with *each* set in the domain of S_i; (2) compute for each $i \in Dom$ the difference set $K_i \setminus \text{lub}(\text{succ}(R,i))$ which contains the irrelevant values and compute the union of all the difference sets; (3) remove from the domain of each successor set S_j such that $j \neq i$, the values which are in the union set.

Example 88 *For $i = 1$, we have* `S1 '< {{1,2,3},{1,4}}` *and the corresponding clique is* `K1 = {{1,2,3},{1,4}, {3,4,5}}`*. The elements removed from the domain of* `S1` *are those in* `K1 \ lub(S1)` *that is the set* `{3,4,5}`*.*

Problem solving One important strength of partial constraint solvers is their dynamic behavior thanks to the delay mechanism. For example the removal of the set `{3,4,5}` from the successor set domains makes it necessary to reconsider the set cardinality constraint over `S3` and `S5` (cf. `appl`). The system infers the two instantiations `S5 = {{4,5}}, S3 = {{1,2,3}}`. From these instantiations, the system activates the `disj_or_eq` constraint and infers: `S1 = S2 = S3 = {{1,2,3}}`, and `S4 = S5  = {{4,5}}`. In this simple example, the optimal and unique solution is found without any labelling procedure. The costs of the various sets does not need to be taken into account.

A larger application has been developed, in which it is necessary to look for an optimal solution using the predicate `min_max/5` and to consider a specific labelling strategy. Both require considering an additional set variable which ranges over a weighted set domain. This domain contains all the sets belonging to Ran with their associated cost. Let `Sw` be this set. The weight constraint `weight(Sw,C)` forms the basis in the minimization process. Additionally, the domain of `Sw` is used in the labelling strategy. The strategy aims at selecting a set among the remaining ones whose costs is the lowest.

The labelling procedure considers each successor set S_i in order. The set `E` with the lowest cost which belongs to `Sw` and to the upper bound of the domain of S_i is selected, and added to S_i. A choice point is created and in case of failure the program backtracks. The previous state is restored and the set E is removed from the domain of S_i.

```
labelling([], _).
labelling([S1 | LSuccs], S) :- set(S1), !,
     labelling(LSuccs, S).
labelling([S1 | LSuccs], S) :-
     lub(S, Lub),
     select_cheapest(S1, E, Lub),
     (E in S1
       ;
     E notin S1),
     labelling([S1 | LSuccs], S).
```

The optimization predicate for the set partitioning problem is:

`min_max((labelling(LSuccs, S), take_min(C)), C, Min, Max, %).`

`take_min(C)` is an integer domain predicate which binds an integer term C to its minimal value. `C` is the weight of the set variable `S`.

To solve the goal `labelling(LSuccs, S), take_min(C)`, we first label all the sets, instantiate the weight of the set domain of `S` to its minimal value and then search for a better solution according to the criteria given.

Computation results A set partitioning problem describing a 0-1 matrix of size 17x197 was implemented using the approach presented here. The complete program takes 4 pages. The problem was taken from the Hoffman and Padberg library [HP92]. The heuristics led to a simplified problem within 7 seconds and the optimal solution was found within 13 seconds. The proof of optimality required 31 additional seconds. The heuristics removed 31 equivalence classes which enables us to divide the number of choice points by 3.

As far as we know this is the first time a set partitioning problem was modelled concisely, and solved with reasonable efficiency within a logic-based language using constraint satisfaction techniques. A modelling using integer domains (0-1) has be tried, but the programmer gave up due to the difficulties he encountered in representing the heuristics.

On the one hand, the flexibility and conciseness of the Conjunto approach is a strength compared with existing mathematical models. On the other hand, constraint satisfaction techniques are not competitive when compared with global methods like the simplex. For example, the system of Padberg et al. dedicated to set partitioning problem solving solves this problem in less than one second. While completing this work, it appeared to us that the set domain CSP approach is promising when investigating feasibility issues that are problematic with the simplex method. The simplex stops when the model is detected to be inconsistent but it cannot detect the reasons for failure. The inherent incremental solving of constraint satisfaction techniques can be of a great help. In addition, the partitioning problem appears as a sub_problem in numerous real life applications (eg. timetables, bus line balancing), which are currently solved using integer domain solvers. While integer domain CSP are well suited to the scheduling constraints of these problems, a set domain CSP can provide an easy way to tackle the partitioning constraints. The cooperation between the solvers is not a problem, provided that the constraints which involve set and integer variables can be attached to both. A real life application is worth considering.

6.3 Efficiency issues: A case study

The previous section illustrated the applicability of the system for dealing with a large class of search problems involving sets, relations, graduations and optimization criteria. The question is: "can a gain in expressiveness be combined with a gain in efficiency ?". From a pruning point of view, the one-to-one correspondence between a set variable ranging over a set domain and a vector of 0-1 variables guarantees that if both sorts of variables are handled using the same labelling procedure (cf. `refine`), the pruning will be exactly the same. If there is a gain, it might therefore come from the saving in memory utilization and consequently from the garbage collection time. This point is illustrated through an integer linear programming optimization problem: the bin packing problem.

Problem description Bin packing problems belong to the class of set partitioning problems [GJ79]. A multiset of n integers $\{w_1, ..., w_n\}$ is given that specifies the weight elements to partition. Another integer W_{max} is given that represents the weight capacity. The aim is to find a partition of the n integers into a minimal number of m bins (or sets) $\{s_1, .., s_k\}$ such that in each bin the sum of all weights does not exceed W_{max}. This problem is usually stated in terms of arithmetic constraints over binary variables and solved using various operations research or constraint satisfaction techniques over binary finite domains. It requires one matrix (a_{ij}) to represent the elements of each set, one vector x_j to represent the selected subsets s_k and one vector w_i to represent the weights of the elements a_{ij}. The cost function to be optimized is the total number of bins.

The mathematical formulation in 0-1 CSP and set domain CSP is described in the following figure.

0-1 CSP abstract formulation	set domain CSP abstract formulation
$\sum_{j=1}^{m} a_{ij}\, x_j = 1$ for all $i \in \{1,..,n\}$	$s_1 \cap s_2 = \{\},\ \ldots, s_{n-1} \cap s_m = \{\}$ $s_1 \cup \ldots \cup s_m = \{(1,w_1),..,(n,w_n)\}$
where: $x_j = 0..1 \begin{cases} 1 & \text{if } s_j \in \{s_1,..,s_k\} \\ 0 & \text{otherwise} \end{cases}$ $a_{ij} = 0..1 \begin{cases} 1 & \text{if } i \in s_j \\ 0 & \text{otherwise} \end{cases}$	$s_j :: \{\}..\{(1,w_1),..,(n,w_n)\}$
$\sum_{i=1}^{n} a_{ij}\, w_i \le W_{max}\ \forall j \in \{1,...,m\}$	$\texttt{weight}(i,w_i) = w_i$ $\sum_{i=1}^{\#glb(s_j)} \texttt{weight}(i,w_i) \le W_{max}\ \forall s_j$

Under these assumptions, the program to solve is to `minimize the number of bins`:

$\texttt{min}\, x_0 = \sum_{j=1}^{m} x_j$	$\texttt{min} x_0 = \#\{s_j \mid s_j \neq \{\}\}$

Problem statement Let $P = \{\ (1,w_1)\,, ..., (i,w_i),\ ..., (n,w_n)\}$ be a non empty set of items i with a weight w_i. The aim is to partition P into a minimal number of N bins such that the sum of the w_i in a computed subset of P does not exceed a limited weight $Wmax$. A bin is represented by a set variable with initial domain $[\{\}, P]$. The union of all bins should be equal to P. This is represented using the `all_union` predicate. All the bins should be pairwise disjoint, which is represented using the `all_disjoint` predicate.

```
pb_statement(N,Max,Sets) :-
    pieces(P),
    make_sets(N,P,Sets),
    state_constraints(Sets,Max,P),

make_sets(0,_Plub,[]).
make_sets(N,Plub,[Set| Sets]):-
    Set `:: [{},Plub],
    N1 is N - 1,
    make_sets(N1,Plub,Sets).
```

```
state_constraints(Sets, P) :-
    restrict_weight(Max,Sets),
    all_disjoint(Sets),
    all_union(Sets,P).

restrict_weight(_M,[]).
restrict_weight(Max,[S| Sets]) :-
    weight(S,W),
    W =< Max,
    restrict_weight(Max,Sets).
```

Problem solving The labelling procedure makes use of the first fit descending heuristic. This heuristic sorts the elements (i, W_i) in decreasing order of their weight. Bins are then filled one after another, which is more efficient than filling all the bins in parallel. The optimization predicate is the classical one for packing problems which initializes the number of bins N to the value $weight(P)/Wmax$ and increases it at each call of goal predicate in case of failure. The solution is the first successful partition. This program was used to solve a large instance of 80 items partitioned into 30 sets. The optimal solution was found in about 22 seconds on a SUN 4/40.

Experimental results and comparisons A comparative study was made with a integer domain (0-1) formulation implemented using the finite domain library of ECL^{i}PS^e. For the encoding of sets and set constraints, we used respectively lists of binary variables and arithmetic constraints on the variables described previously. The arithmetic constraint predicates were handled using the ECLiPSe solver[1] of arithmetic constraints over finite domains. The 0-1 integer domain program was encoded so as to use the same first fit descending heuristics and the same labelling procedure as the set domain CSP program. The following array gives the results regarding time consumption together with space utilization.

Criterion	Conjunto program	FD program
global stack peak (bytes)	847 872	2 334 720
trail stack peak (bytes)	126 968	987 136
garb. collection number	27	77
cpu time (sec.)	21.6	31.5
garb. collection time (sec.)	1.21	6.28

The two programs differ in the data structure used, and thus in the constraints applied to these data. The first point to note is that this difference has an impact both on the space usage (stack peaks[2]) and on the cpu time. The space utilization comprises, among other stacks, the global stack and the trail stack. The data structure is largely responsible for the growth of the global stack peak. The difference in space utilization (stack sizes) between the two approaches comes from the set-like representation as a list of zero-one domain variables versus two sorted lists in Conjunto. The lists of zero-one variables are never reduced because

[1] based on consistency techniques which perform a reasoning about variation domain bounds or about variation domains, depending on the constraint predicate.

[2] the peak value indicates what the maximum amount allocated was during the session.

retrieving an element from a set corresponds to setting a variable domain to zero. This is not the case with the set domain representation.

The trail stack is used to record information (set domains or lists of zero-one variables) that is needed on backtracking. The number of times the two program execution backtrack is the same, so the difference comes from the amount of information recorded.

The garbage collection number is the times garbage collections are performed which is closely linked to the global and trail stack because the garbage collection on both at the same time. Thus, the difference in the garbage collection number comes again from the space utilization.

The difference between the cpu times is due first to the time needed for garbage collection which is a direct consequence of the size of the global and trail stack; and secondly to the time needed for performing operations on the data.

Profiling the cpu time consumption indicates that half of time spent in the FD program resolution is the time needed for performing arithmetic operations on the zero-one variables. The weight constraint applied to each set is one of the most expensive computations. The weight constraint $a_{i1} \times w_1 + a_{i2} \times w_2 + \ldots a_{in} \times w_n \leq w_{max}$ which is woken each time an a_{ij} is set to 1, consists of a Cartesian product of two lists. In the Conjunto program, it consists in constraining the sum of weights w_i directly available from the elements (i, w_i) of a domain upper bound. Another costly computation in the FD formulation, is the computation of the largest weight not already considered for one set. This requires checking the value of the zero-one variable, and if this value is one, considering the weight associated to this variable. A weight is not considered if the corresponding domain variable is not instantiated. In the Conjunto program, this computation corresponds to the difference of the two bounds of a set domain, and the resulting set contains the elements $(i,\, w_i)$ which have not yet been considered. Computing this difference is in fact the most time consuming step in the Conjunto program resolution, because it is also performed when computing disjoint sets, but it represents half of the cpu time consumption of arithmetic operations.

This application shows that set constraints together with set domains are expressive enough to embed the problem semantics, and to avoid encoding the information as lists of binary variables or handling additional data (the list of weights). It also shows that consistency techniques for set constraints are efficient enough to solve combinatorial problems on sets.

6.4 Conclusion

In this chapter, we have shown how set based combinatorial search problems coming from combinatorial mathematics and operations research can be modelled and solved using Conjunto. The modelling is based on a set domain CSP approach and the solving on constraint satisfaction and search techniques. The solving of set-based optimization problems is possible thanks to the graduated constraints (set cardinality and weight constraints) which map set terms onto quantifiable terms.

With regard to an integer domain CSP, a set domain CSP approach contributes transparency with respect to the mathematical definition of set problems, and allows the user to go from a hypergraph to a graph representation, thus reducing the number of variables and simplifying the constraint statement phase. As far as efficiency is concerned, the first application (ternary Steiner problem) showed that the solving of set constraint achieves a pruning identical to that of global constraints. The cpu were also similar. This can be generalized to the class of global constraints whose behaviour resembles that of set constraints. The second application (bin packing) showed that an efficient set labelling procedure in a set domain CSP, provides a pruning equivalent to the one of the labelling procedure currently used for 0-1 CSP problems i.e., assigning one by one to 0-1 variables from a boolean vector the value 1 (or 0 in case of failure). Consequently, any 0-1 CSP can be modelled more concisely using Conjunto with a possible gain in efficiency. The gain comes essentially from the time needed for garbage collection which is more important in the 0-1 CSP approach which uses a larger amount of variables.

The last application (set partitioning) makes us of the one-to-one correspondence between a set variable ranging over a set domain and a 0-1 vector which allows us to model 0-1 Integer Linear Programming (ILP) problems as set domain CSPs. The modelling of 0-1 ILP problems as set domain CSPs in a constraint logic programming language shows the programming facilities of logic programming and enhances the class of CSPs. In particular, a CSP view of 0-1 ILPs brings flexibility to the modelling and can be useful when (1) unpure 0-1 ILP problems are to be tackled, (2) when their feasibility is problematical with ILP tools, (3) and when small 0-1 ILP problems are involved in some real CSP applications (eg. timetables, bus line balancing, etc).

Conclusion

Que chaque critique t'élève,
car tes possibilités s'élargissent avec elle !

Du matin au soir,
ne cesse pas d'appeler le Nouveau.

In this document, we have described the formal and practical framework of a new constraint logic programming language over sets. Its design and implementation allowed us to tackle efficiently set-based combinatorial search problems with a natural and concise modelling. The word "natural" is referring to the transparency of the modelling with respect to the mathematical formulation of the problem. The language models set-based problems as set domain Constraint Satisfaction Problems (CSP), and solves them using constraint satisfaction techniques. On the one hand, the set domain CSP paradigm extends the standard CSP paradigm to deal with partially ordered domains. On the other hand, we do not lose the pruning power of constraint satisfaction techniques when applying them over set and graduated constraints. The applications developed with the Conjunto language showed its practical viability.

Today, the Conjunto solver is available as a library in the ECL^{i}PS^e platform, developed at ECRC. An industrial interest for this solver has appeared while we were implementing the system. Set constraints over set domains are now embedded in the ILOG solver.

While our work has essentially aimed at solving applications, it has provided us with a matter for a formal definition of the language. The formal framework distinguishes between the computation domain of the constraint logic programming language, and the constraint domain over which the computations are actually performed. These two levels of discourse are linked together by approximations and closure operations. On the one hand, the user reasons on elements from the computation domain. On the other hand, the constraint solver performs computations over elements from the constraint domain. Up to now, CLP(FD) languages are defined as constraint logic programming languages, but their formal definition is still based on the formal framework defined by Van Hentenryck that is, embedding consistency techniques in logic programming. The formal description of the Conjunto language can be used to give a formal definition of CLP(FD) languages in the CLP framework, since both systems handle constraints in a similar way.

The applications that we have considered are operations research and combinatorial mathematics problems. However, those lasts years the notions of set constraints and set domains have been set for other purposes as well.

Related work

A related line of work is program analysis systems [HJ91] [AW92] [BGW93] [Aik94] among others. They handle a class of sets (possibly infinite sets) larger than that of CLP(Sets) languages or Conjunto, and deal with set constraints of the form $s \subseteq s_1$ where s and s_1 denote specific set expressions (depending on the system at hand). The different resolution algorithms are based on transformation algorithms which preserve the consistency of the system either by computing a least model [HJ91] which does not preserve all solutions, or by computing a finite set of systems in solved form [AW92]. In [BGW93], the authors demonstrated that the latter algorithm takes non-deterministic exponential time. The difference between these systems and the class of CLP(Sets) languages is that they do not interpret set operations. However, they show the expressiveness of set constraints for the analysis of programs developed in logic programming, functional programming, etc.

Another line of research which has some similar points with set domains is the rough set theory. Rough sets have been introduced in [Paw84] [Paw91] as a tool for dealing with incomplete knowledge in applications from artificial intelligence (decision systems, pattern recognition, approximate reasoning, etc.). In order to reason on imprecise data in an information system, rough sets approximate the data by a pair of sets similar to the set domain concept. The idea consists in representing an information system as a data table which contains partial information about some objects in terms of attribute values. The row indices of the data table contain the set of objects and the column indices, the list of attributes. The attribute values intersect rows and columns to describe the partial information which characterizes the objects. In general, any pair of objects in an information system may have identical values for some attributes. Such similarities among objects are reflected by a relation called the "indiscernability relation". It is an equivalence class over the sets of objects (called the universe). This relation is used to define approximations of sets of objects from the universe. Two types of approximations are defined, the lower and upper approximations. Each of these approximations tells us whether a set of object can be characterized by a given set of attributes. The lower approximation contains the set of objects which can be definitely characterized by the attributes and the upper bound contain the set of objects which might be characterized by the attributes. If some objects being in the upper bound do not appear in the lower one, this means that they are described by the same attribute values, and consequently can not be characterized by this set of attributes. The concept of rough sets differs from that of set domains essentially in two points. On the one hand, rough sets derive approximations from an external parameter which is the class of attributes considered. On the other hand, the approximations are not used to search for variable values, but to answer the following questions. If a set of objects can not be characterized in an information system can it be approximately characterized ? Is the whole knowledge necessary to describe an information system ? To which extent can we reduce it while keeping the initial information ?

Further developments

Some issues are still open with respect to "what we did not do and remains to be done". We believe that some further research on applications and algorithms is needed.

Applications The concept of graduated constraints helped us with tackling set-based optimization problems, and studying the cooperation between two solvers (Conjunto and integer domain solvers), but the search space was defined with set domains essentially. The Conjunto language has not been used so far to tackle real life applications defined over a search space containing also integer domains. Applications involving scheduling constraints and set constraints are still to be developed. In particular, they would allow us to figure out whether it is possible or not to work on a mixed-search space. Time tables, bus line balancing, are some of the applications.

Another point that has not been considered yet, is the use of the language to deal with other application domains like databases. In recent years, linear constraints and constraint solving techniques over tuples of relations have been respectively embedded in constraint databases and query languages. The main motivations are respectively (1) to use constraints to model an infinite number of relations, (2) to use consistency techniques (mainly forward checking) for query optimization. The former approach (see [KKR90]) considers linear constraints to model some classes of databases (e.g. in graphics). In the latter approach, a constraint in a database query is a condition that must be satisfied by answers to the query (see [WBP95]). One can think of using set constraints in the former approach to model other sorts of databases. In the second approach set constraints could be used to state queries over collections of tuples.

Algorithms Regarding the class of consistency methods we have been using, we have essentially considered node and arc consistency techniques applied to set and graduated constraints. It sounds interesting to go beyond this, to use path consistency algorithms, and to take into account the latest researchs on the topology of constraint graphs. Some issues might be different from those already established with respect to integer domain CSPs. In this respect, the study of the ratio complexity/pruning is very important.

Future work

More work has to be done on extending the class of graduated constraints. Currently they map set domains to integer domains, that is a partially ordered structure to an ordered one. It could be interesting to consider mappings on two partially ordered structures, for example from sets to real intervals or vice versa. This would extend the expressivity and the application domain of the language. This requires studying the formal properties of such mappings and the nature of their closure which deal with elements from a powerset of convex parts. It also requires studying their handling when using constraint satisfaction techniques, in particular the degree of pruning achieved during the resolution is an important issue with respect to a practical use of these mappings.

It would also be interesting to extend the set domain concept to that of lattice domains. When solving set partitioning and Steiner problems we realized that if lattice domains and lattice inclusion constraints had been provided, the handling of a set of equivalence classes in the partitioning problem would have been easier. For example, considering the lattice domains $\{\{1,3\},\{1,2\}\}$ and $\{\{1,2,3\}\}$, we have $\{\{1,3\},\{1,2\}\} \sqsubseteq \{\{1,2,3\}\}$. In addition, the global reasoning on the Steiner problem can be achieved in a straightforward way. A solution to the ternary Steiner problem modelled with lattice domains and constraints would have been the value of a single lattice variable, and consequently the symmetries generated by possible permutations of triples disappear. A set of constraints applied to variables ranging over lattice domains would ease the modelling and solving of set based problems dealing with the search for equivalence classes. They would model a set domain CSP as a lattice domain CSP, and thus add a higher level of expressiveness with respect to set domains. On the one hand, the formal framework corresponding to embedding lattice intervals in CLP can be derived from the one we have presented. On the other hand, the practical framework requires further works describing the algorithms and studying the trade-off between expressiveness and efficiency.

Annexe A
The set domain library: user manual

We present the user manual of the set domain library which is currently available in ECL^{i}PS^e. It does not comprise the mapping terms and constraints.

Conjunto is a system to solve set constraints over finite set domain terms. It has been developed using the kernel of ECL^{i}PS^{e}based on metaterms (attributed variables). It contains the finite domain library of ECL^{i}PS^e. The library **conjunto.pl** implements constraints over set domain terms that contain herbrand terms as well as ground sets. Modules that use the library must start with the directive

`:- use_module(library(conjunto))` or `:- lib(conjunto)`.

For those who are already familiar with the ECL^{i}PS^{e}extension manual this manual follows the finite domain library structure.

Note: for any question or information request, please send an email to carmen@ecrc.de.

A.1 Syntax

- A ground set is written using the characters { and }, e.g. $S = \{1, 3, \{a, g\}, f(2)\}$
- A domain D attached to a set variable is specified by two ground sets : $[Glb_s, Lub_s]$
- Set expressions: Unfortunately the characters representing the usual set operators are not available on our monitors so we use a specific syntax making the connection with arithmetic operators:
  - $\cup$ is represented by `\/`,
  - $\cap$ is represented by `/\`,
  - $\setminus$ is represented by `\`

A.2 The solver

The **Conjunto** solver acts in a data driven way using a relation between *states*. The transformation performs interval reduction over the set domain bounds. The set expression domains are approximated in terms of the domains of the set variables involved. From a constraint propagation viewpoint this means that constraints over set expressions can be approximated in terms of constraints over set variables. A failure is detected in the constraint propagation phase as soon as one domain lower bound glb_s is not included in its associated upper bound lub_s. Once a solved form has been reached all the constraints which are not definitely solved are delayed and attached to the concerned set variables.

A.3 Constraint predicates

?Svar ‘:: [++Glb,++Lub]

attaches a domain to the set variable or to a list of set variables *Svar*. If $Glb \not\subseteq Lub$ it fails. If $Svar$ is already a domain variable its domain will be updated according to the new domain; if $Svar$ is instantiated it fails. Otherwise if $Svar$ is free it becomes a set variable.

set(?Term)

succeeds if *Term* is a ground set.

?S ‘= ?S1

The value of the set term S is equal to the value of the set term *S1*.

?E `in` **?S**

The element E is an element of S. If E is ground it is added to the lower bound of the domain of S, otherwise the constraint is delayed. If E is ground and does not belong to the upper bound of S domain, it fails.

?E `notin` **?S**

The element E does not belong to S. If E is ground it is removed from the upper bound of S, otherwise the constraint is delayed. If E

is ground and belongs to the upper bound of the domain of S, it is removed from the upper bound and the constraint is solved. If E is ground and belongs to the lower bound of S domain, it fails.

?S `< ?S1

The value of the set term S is a subset of the value of the set term **S1**. If the two terms are ground sets it just checks the inclusion and succeeds or fails. If the lower bound of the domain of S is not included in the upper bound of *S1* domain, it fails. Otherwise it checks the inclusion over the bounds. The constraint is then delayed.

?S `<> ?S1

The domains of S and S1 are disjoint (intersection empty).

`all_union`**(?Lsets, ?S)**

Lsets is a list of set variables or ground sets. S is a set term which is the union of all these sets. If S is a free variable, it becomes a set variable and its attached domain is defined from the union of the domains or ground sets in *Lsets*.

`all_disjoint`**(?Lsets)**

Lsets is a list of set variables of ground sets. All the sets are pairwise disjoint.

#(?S,?C)

S is a set term and C its cardinality. C can be a free variable, a finite domain variable or an integer. If C is free, this predicate is a mean to access the set cardinality and attach it to C. If not, the cardinality of S is constrained to be C.

`weight`**(?S,?W)**

S is a set variable whose domain is a *weighted domain*. W is the weight of S. If W is a free variable, this predicate is a mean to access the set weight and attach it to W. If not, the weight of S is constrained to be W. e.g.

$$S` :: [\{(2,3)\}, \{(2,3),(1,4)\}], \mathtt{weight}(S, W)$$

returns W :: 3..7

refine(?Svar)

If *Svar* is a set variable, it labels *Svar* to its first possible domain value. If there are several instances of *Svar*, it creates choice points. If *Svar* is a ground set, nothing happens. Otherwise it fails.

A.4 Examples

A.4.1 Set domains and interval reasoning

First we give a very simple example to demonstrate the expressiveness of set constraints and the propagation mechanism.

```
:- lib(conjunto).

[eclipse 2]: Car `:: [{renault}, {renault,bmw,mercedes,peugeot},
             Type_french = {renault,peugeot} ,
             Choice `= Car /\ Type_french.

Choice = Choice{[{renault}, {peugeot, renault}]}
Car = Car{[{renault}, {bmw, mercedes, peugeot, renault}]}
Type_french = {peugeot,renault}

Delayed goals:
        inter_s({peugeot, renault}, Car{[{renault},{bmw,mercedes,
                 peugeot, renault}]}
        Choice{[{renault}, {peugeot, renault}]})
yes.
```

If now we add one cardinality constraint:

```
[eclipse 3]: Car `:: [{renault}, {renault, bmw, mercedes,peugeot}],
             Type_french = {renault, peugeot} ,
             Choice `= Car /\ Type_french,
             #(Choice, 2).

Car = Car{[{peugeot, renault}, {bmw, mercedes, peugeot, renault}]}
Type_french = {peugeot, renault}
Choice = {peugeot, renault}

yes.
```

The first example gives a set of cars from which we know `renault` belongs to. The other labels `{renault, bmw, mercedes, peugeot}` are possible elements of this set. The `Type_french` set is ground and `Choice` is the set term resulting from the intersection of the first two sets. The first execution tells us that `renault` is element of `Choice` and `peugeot` might be one. The intersection constraint is partially satisfied and might be reconsidered if one of the domain of the set terms involved changes. The cosntraint is delayed.

In the second example an additional constraint restricts the cardinality of `Choice` to 2. Satisfying this constraint implies setting the `Choice` set to `{peugeot, renault}`. The domain of this set has been modified so is the intersection constraint activated and solved again. The final result adds `peugeot` to the `Car` set variable. The intersection constraint is now satisfied and removed from the constraint store.

A.4.2 Subset-sum computation with convergent weight

A more elaborate example is a small decision problem. We are given a finite weighted set and a *target* $t \in N$. We ask whether there is a subset s' of S whose weight is t. This also corresponds to having a single weighted set domain and to look for its value such that its weight is t.

This problem is NP-complete. It is approximated in Integer Programming using a procedure which "trims" a list according to a given parameter. For example, the set variable S `:: $[\{\}, \{(a, 104), (b, 102), (c, 201), (d, 101)\}]$ is approximated by the set variable $S'$ `:: $[\{\}, \{(c, 201), (d, 101)\}]$ if the parameter delta is 0.04 ($0.04 = 0.2 \div n$ where $n = \#S$).

```
:- lib(conjunto).
```

```
%Find the optimal solution to the subset-sum problem
solve(S1, Sum) :-
     getset(S),
     S1 `:: [{}, S],
     trim(S, S1),
     constrain_weight(S1, Sum),
     weight(S1, W),
     Cost = Sum - W,
     min_max(labelling(S1), Cost).

%The set weight has to be less than Sum
constrain_weight(S1, Sum) :-
     weight(S1, W),
     W #<= Sum.

%Get rid of a set of elements of the set according to a given delta
trim(S, S1) :-
     set2list(S, LS),
     trim1(LS, S1).

trim1([E | LS], S1) :-
     getdelta(D),
     testsubsumed(D, E, LS, S1).

testsubsumed(_, _, [], _).
testsubsumed(D, E, [F | LS], S1) :-
     el_weight(E, We),
     el_weight(F, Wf),
     (We =< (1 - D)*Wf
        ->
       testsubsumed(D, F, LS, S1);
       F notin S1,
    testsubsumed(D, E, LS, S1)).

%Instantiation procedure
labelling(Sub) :-
     set(Sub),!.
labelling(Sub) :-
     max_weight(Sub, X),
     (X in Sub
```

```
        ;
       X notin Sub),
     labelling(Sub).

%Some sample data
getset(S) :-
     S = {(a,104), (b,102), (c,201) ,(d,101), (e,305), (f,50),
          (g,70),(h,102)}.
getdelta(0.05).
```

The approach is the following: first create the set domain variable(s), here there is only one which is the set we want to find. We state constraints which limit the weight of the set. We apply the "trim" heuristics which removes possible elements of the set domain. And finally we define the cost term as a finite domain used in the `min_max/2` predicate. The cost term is an integer. The **conjunto.pl** library makes sure that any modification of an fd term involved with a set term is propagated on the set domain. The labelling procedure refines a set domain by selecting the element of the set domain which has the biggest weight using `max_weight(Sub, X)`, and by adding it to the lower bound of the set domain. When running the example, we get the following result:

```
[eclipse 3]: solve(S, 550).

Found a solution with cost 44
Found a solution with cost 24
10 backtracks
0.116667
S = {(f, 50), (g, 70), (c, 101), (e, 305)}
yes.
```

An interesting point is that in set based problems, the optimization criteria mainly concern the cardinality or the weight of a set term. So in practice we just need to label the set term while applying the **fd** optimization predicates upon the set cardinality or the set weight. There is no need to define additional optimization predicates.

A.5 When to use set variables and constraints...

The *subset-sum* example shows that the general principle of solving problems using set domain constraints works just like finite domains:

- Stating the variables and assigning an initial set domain to them.
- Constraining the variables. In the above example the constraint is just a built-in constraint but usually one needs to define additional constraints.
- Labelling the variables, *i.e.*, assigning values to them. In the set case it would not be very efficient to select one value for a set variable for the size of a set domain is exponential in the upper bound cardinality and thus the number of backtracks could be exponential too. A second reason is that no specific information can be deduced from a failure (backtrack) whereas if (like in the refine predicate) we add one by one elements to the set till it becomes ground or some failure is detected, we benefit much more from the constraint propagation mechanism. Every domain modification activates some constraints associated to the variable (depending on the modified bound) and modifications are propagated to the other variables involved in the constraints. The search space is then reduced and either the goal succeeds or it fails. In case of failure the labelling procedure backtracks and removes the last element added to the set variable and tries to instanciate the variable by adding another element to its lower bound. In the `subset-sum` example the labelling only concerns a single set. Although the choice for the element to be added can be done without specific criterion like in the `steiner` example, some user defined heuristics can be embedded in the labelling procedure like in the `subset-sum` example. Then the user needs to define his own `refine` procedure.

Set constraints propose a new modelling of already solved problems or allows (like for the *subset-sum* example) to solve new problems using CLP. Therefore, one should take into account the problem semantics in order to define the initial search space as small as possible and to make a powerful use of set constraints. The objective of this library is to bring CLP to bear on graph-theorical problems, thus leading to a better specification and solving of problems as, packing and partitioning which find their application in many real life problems. A partial list includes: railroad crew scheduling, truck deliveries, airline crew scheduling, tanker-routing, information retrieval,time tabling problems, location problems, assembly line balancing, political districting,etc.

Sets seem adequate for problems where one is not interested in each element as a specific individual but in a collection of elements where no specific distinction is made and thus where symmetries among the element values need to be avoided (eg. steiner problem). They are also useful when heterogeneous constraints are involved in the problem like weight constraints combined with some disjointness constraints.

A.6 User-defined constraints

To define constraints based on set domains one needs to access the properties of a set term like its domain, its cardinality, its possible weight. As the set variable is a metaterm i.e. an abstract data structure, some built-in predicates allow the user to process the set variables and their domains, modify them and write new constraint predicates.

A.6.1 The abstract set data structure

A set domain variable is a metaterm. The **conjunto.pl** library defines a metaterm attribute

set with [setdom : [Glb,Lub], card: C, weight: W, del_glb: Dglb, del_lub: Dlub, del_any: Dany]

This attribute stores information regarding the set domain, its cardinality, and weight (null if undefined) and together with three suspension lists. The attribute arguments have the following meaning:

- **setdom** The representation of the domain itself. As set domains are treated as abstract data types, the users should not access them directly, but only using built-in access and modification predicates presented hereafter.
- **card** The representation of the set cardinality. The cardinality is initialized as soon as a set domain is attached to a set variable. It is either a finite domain or an integer. It can be accessed and modified in the same way as set domains (using specific built-in predicates).
- **weight** The representation of the set weight. The weight is intialized to zero if the domain is not a weighted set domain, otherwise it is computed as soon as a weighted set domain is attached to a set variable. it can be accessed and modified in the same way as set domains (using specific built-in predicates).

- **del_glb** A suspension list that should be woken when the lower bound of the set domain is updated.
- **del_lub** a suspension list that should be woken when the upper bound of the set domain is updated.
- **del_any** a suspension list that should be woken when any reduction of the domain is inferred.

The attributes of a set domain variable can be accessed with the predicate **svar_attribute/2** or by unification in a matching clause:

```
get_attribute(_{set: Attr}, A) :-
      -?->
      nonvar(Attr),
      Attr = A.
```

The attribute arguments can be accessed by macros from the ECL^{i}PS^e **structures.pl** library, if e.g. **Attr** is the attribute of a set domain variable, the del_glb list can be obtained by:

arg(del_glb of set, Attr, Dglb)

or by using a unification:

Attr = set with del_glb: Dglb

A.6.2 Set Domain access

The domains are represented as abstract data types, and the users are not supposed to access them directly. So we provide a number of predicates to allow operations on set domains.

set_range(?Svar,?Glb,?Lub)

If *Svar* is a set domain variable, it returns the lower and upper bounds of its domain. Otherwise it fails.

glb(?Svar,?Glb)

If *Svar* is a set domain variable, it returns the lower bound of its domain. Otherwise it fails.

lub(?Svar, ?Lub)

> If *Svar* is a set domain variable, it returns the upper bound of its domain. Otherwise it fails.

el_weight(++E, ?We)

> If *E* is element of a weighted domain, it returns the weight associated to *E*. Otherwise it fails.

max_weight(?Svar,?E)

> If *Svar* is a set variable, it returns the element of its domain which belongs to the set resulting from the difference of the upper bound and the lower bound and which has the greatest weight. If *Svar* is a ground set, it returns the element with the biggest weight. Otherwise it fails.

Two specific predicates make a link between a ground set and a list.

set2list(++S, ?L)

> If *S* is a ground set, it returns the corresponding list. If *L* is also ground it checks if it is the corresponding list. If not, or if *S* is not ground, it fails.

list2set(++L, ?S)

> If *L* is a ground list, it returns the corresponding set. If *S* is also ground it checks if it is the corresponding set. If not, or if *L* is not ground, it fails.

A.6.3 Set variable modification

A specific predicate operate on the set domain *variables.*

When a set domain is reduced, some suspension lists have to be scheduled and woken depending on the bound modified.

NOTE: There are 3 suspension lists in the **conjunto.pl** library, which are woken precisely when the event associated with each list occurs. For example, if the lower bound of a set variable is modified, two suspension lists will

be woken: the one associated to a **glb** modification and the one associated to **any** modification. This allows user-defined constraints to be handled efficiently.
modify_bound(Ind, ?S, ++Newbound)

> **Ind** is a flag which should take the value **lub** or **glb**, otherwise it fails ! If S is a ground set, it succeeds if we have *Newbound* equal to S. If S is a set variable, its new lower or upper bound will be updated. For monotonicity reasons, domains can only get reduced. So a new upper bound has to be contained in the old one and a new lower bound has to contain the old one. Otherwise it fails.

A.7 Example of defining a new constraint

The following example demonstrates how to create a new set constraint. To show that set inclusion is not restricted to ground herbrand terms we can take the following contraint which defines lattice inclusion over lattice domains:

$$S_1 \texttt{incl}\ S$$

Assuming that S and S_1 are specific set variables of the form $S` :: [\{\}, \{\{a, b, c\}, \{d, e, f\}\}]$, $S_1` :: [\{\}, \{\{c\}, \{d, f\}, \{g, f\}\}]$, we would like to define such a predicate that will be woken as soon as one or both set variables' domains are updated in such a way that would require updating the other variable's domain by propagating the constraint. This constraint definition also shows that if one wants to iterate over a ground set (set of known elements) the transformation to a list is convenient. In fact iterations do not suit sets and benefit much more from a list structure. We define the predicate `incl(S,S1)` which corresponds to the following program. The program is quite long. Extending the solver to bear on lattice domains and constraints over lattices would add a lot of expressivity.

```
:- use_module(library(conjunto)).

incl(S,S1) :-
      set(S),set(S1),
      !,
      check_incl(S, S1).
incl(S, S1) :-
      set(S),
      set_range(S1, Glb1, Lub1),
      !,
```

```
      check_incl(S, Lub1),
      S \/ Glb1 `= S1NewGlb,
      modify_bound(glb, S1, S1NewGlb).
incl(S, S1) :-
      set(S1),
      set_range(S, Glb, Lub),
      !,
      check_incl(Glb, S1),
      large_inter(S1, Lub, SNewLub),
      modify_bound(lub, S, SNewLub).
incl(S,S1) :-
      set_range(S, Glb, Lub),
      set_range(S1, Glb1, Lub1),
      check_incl(Glb, Lub1),
      Glb \/ Glb1 `= S1NewGlb,
      large_inter(Lub, Lub1, SNewLub),
      modify_bound(glb, S1, S1NewGlb),
      modify_bound(lub, S, SNewLub),
      ((set(S); set(S1))
         -> true
            ;
       make_suspension(incl(S, S1),2, Susp),
       insert_suspension([S,S1], Susp, del_any of set, set)
      ),
     wake.

large_inter(Lub, Lub1, NewLub) :-
      set2list(Lub, Llub),
      set2list(Lub1, Llub1),
      largeinter(Llub, Llub1, LNewLub),
      list2set(LNewLub, NewLub).
largeinter([], _, []).
largeinter([S | List_set], Lub1, Snew) :-
      largeinter(List_set, Lub1, Snew1),
      (contained(S, Lub1)
         -> Snew = [S | Snew1]
            ;
            Snew = Snew1
      ).

check_incl({}, _S) :-!.
```

```
check_incl(Glb, Lub1) :-
      set2list(Glb, Lsets),
      set2list(Lub1, Lsets1),
      all_union(Lsets, Union),
      all_union(Lsets1, Union1),
      Union < Union1,!,
      checkincl(Lsets,Lsets1).

checkincl([], _Lsets1).
checkincl([S | Lsets],Lsets1):-
     contained(S, Lsets1),
     checkincl(Lsets,Lsets1).

contained(_S, []) :- fail,!.
contained(S, [Ss | Lsets1]) :-
    (S < Ss -> true
                ;
               contained(S, Lsets1)
    ).
```

The execution of this constraint is dynamic, *i.e.*, the predicate `incl/2` is called and woken following the following steps:

- We check if the two set variables are ground `set`. If so we just check deterministically if the first one is included (lattice inclusion) in the second one `check_incl`. This predicate checks that any element of a ground set (which is a set itself in this case) is a subset of at least one element of the second set. If not it fails.

- We check if the first set is ground and the second is a set domain variable. If so, `check_incl` is called over the first ground set and the upper bound of the second set. If it succeeds, then the lower bound of the set variable might not be consistent any more, we compute the new lower bound (*i.e.*, adding elements from the ground set in it (by using the union predicate) and we modify the bound `modify_bound`. This predicate also wakes all concerned suspension lists and instantiates the set variable if its domain is reduced to a single set (upper bound = lower bound).

- We check if the second set is ground and the first one is a set variable. If so, `check_incl` is called over the lower bound of the first set and the second ground set. If it succeeds then the upper bound of the set variable might not be consistent any more. The new upper bound is computed by intersecting the first set with the upper bound of the set variable in the lattice acceptation `large_inter` and is updated `modify_bound`.

- we check if both set variables are domain variables. If so the lower bound of the first set should be included in the lattice sense in the upper bound of the second one `check_incl`. If it succeeds, then if the lower bound the second set is no more consistent we compute the new one by making the union with first sec lower bound. In the same way, the upper bound of the first set might not be consistent any more. If so, we compute the new one by intersecting (in the lattice acceptation) the both upper bounds to compute the new upper bound of the first set `large_inter`. The upper bound of the first set variable is updated as well as the lower bound of the second set `modify_bound`.

- After checking all these updates, we test if the constraint implies an instanciation of one of the two sets. If this is not the case, we have to suspend the predicate so that it is woken as soon as any bound of either set domain is changed. The predicate `make_suspension/3` can be used for any ECL^{i}PS^{e}module based on a meta-term structure. It creates a suspension, and then the predicate `insert_suspension/4`, puts this suspension into the appropriate lists (woken when any bound is updated) of both set variables.

- the last action `wake` triggers the execution of all goals that are waiting for the updates we have made. These goals should be woken after inserting the new suspension, otherwise the new updates coming from these woken goals won't be propagated on this constraint !

A.8 Set Domain output

The library **conjunto.pl** contains output macros which print a set variable as well as a ground set respectively as an interval of sets or a set. The **setdom** attribute of a set domain variable (metaterm) is printed in the simplified form of just the $[glb, lub]$ interval, e.g.

```
[eclipse 2]: S `:: [{},{a,v,c}], svar_attribute(S,A),
             A = set with setdom : D.

S = S{[{}, {a,c,v}]}
A = [{}, {a,c,v}]
D = [{}, {a, c, v}]
yes.
```

A.9 Debugger

The ECL^{i}PS^e debugger which supports debugging and tracing of finite domain programs in various ways, can just be used the same way for set domain programs. No specific set domain debugger has been implemented for this release.

Index

Bibliography

[Aik94] A. Aiken. Set Constraints: Results, Applications and Future Directions. In *PPCP'94*. Principle and Practice of Constraint Programming, 1994.

[AW92] Alexander Aiken and Edward L. Wimmers. Solving Systems of Set Constraints. In *IEEE Symposium on Logic in Computer Science*, June 1992.

[BC94] N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. In Elsevier Science, editor, *Mathematical Computation Modelling*, volume 20(12), pages 97–123. Pergamon, 1994.

[BDPR94] P. Bruscoli, A. Dovier, E. Pontelli, and G. Rossi. Compiling Intensional Sets in CLP. In P. Van Hentenryck, editor, *ICLP'94*, pages 647–664, 1994.

[Bel90a] N. Beldiceanu. Definition of Global Constraints. Internal Report IR-LP-22-30, ECRC, Munich Germany, Dec 1990.

[Bel90b] N. Beldiceanu. An example of introduction of global constraints in CHIP: Application to block theory problems. Technical Report TR-LP-49, ECRC, Munich Germany, May 1990.

[Ben95] F. Benhamou. Interval Constraint Logic Programming. In A. Podelski, editor, *Constraint Programming: Basics and Trends*. LNCS, Springer Verlag, 1995. to appear.

[BGW93] L. Bachmair, H. Ganzinger, and U. Waldmann. Set Constraints are the Monadic Class. In *Proceedings of the LICS'93*, 1993.

[Bir67] G. Birkhoff. *Lattice Theory*, volume 25 of *Colloquium Publications*. American Mathematical Society, Providence, RI, 1967. Chapter I.

[BM70] M. Barbut and B. Montjardet. *Ordre et Classification: algèbre et combinatoire 1*, volume 1. Hachette, 1970. French.

[BMH94] F. Benhamou, D. MacAllester, and P. Van Hentenryck. CLP (Intervals) revisited. In *ILPS'94*, pages 124–138, Ithaca, NY, USA, 1994.

[BNST91] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set constructors in a logic database language. In *Journal of Logic Programming*, pages 181–232. Elsevier, New-York, 1991.

[BT95] F. Benhamou and Touraivane. Prolog IV: langage et algorithmes. In *Journées francophones de la programmation logique*, pages 50–64. JFPL'95, 1995. in French.

[Bun84] A. Bundy. A generalized Interval Package and its use for Semantical Checking. In *ACM Transaction on Mathematical Systems*, chapter 10 (4), pages 397–409. 1984.

[CKC83] A. Colmerauer, H. Kanoui, and M. Van Caneghem. Prolog, bases théoriques et développements actuels. *T.S.I. (Techniques et Sciences Informatiques)*, 2(4):271–311, 1983.

[CKPR73] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un système de communication homme-machine en Français. tech. rep., AI Group, Université d'Aix-Marseille II, 1973.

[Cle87] J.G. Cleary. Logical arithmetic. In *Future Generation Computing Systems*, chapter 2(2), pages 125–149. 1987.

[Coh90] J. Cohen. Constraint Logic Programming Languages. *Communications of the ACM*, 33(7):52–61, July 1990.

[Col87] A. Colmerauer. Opening the prolog III Universe. In *BYTE magazine*. 1987.

[Col90] A. Colmerauer. An introduction to Prolog III. *Communications of ACM*, 33(7):70–90, July 1990.

[CP94] Y. Caseau and Jean-F. Puget. Constraints on Order-Sorted Domains. In *ECAI'94*, 1994.

[DHS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and F. Graf. Applications of CHIP to industrial and engineering problems. *Artificial Intelligence and Expert Systems*, June 1988.

[DOPR91] A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. {log}: A Logic Programming Language with Finite Sets. In *ICLP'91*, pages 111–124, Paris, June 1991.

[DR93] A. Dovier and G. Rossi. Embedding Extensional Finite Sets in CLP. In *ILPS'93*, 1993.

[DSea88] M. Dincbas, H. Simonis, and P. Van Hentenryck et al. The Constraint Logic Programming Language CHIP. In *FGCS*, Japan, Aug. 1988.

[DSH88a] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 1988.

[DSH88b] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car-Sequencing Problem in Constraint Logic Programming. In *ECAI*, Munich, Aug. 1988.

[ECR94] ECRC. ECLiPSe (a) user manual, (b) extensions of the user manual. Technical report, ECRC, Jan 1994.

[Fik70] R. E. Fikes. Ref-arf: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[Flo67] R. W. Floyd. Nondeterministic algorithms. *Journal of the Association for Computing Machinery*, 14(4):636–644, Oct 1967.

[Fra86] R. Fraissé. *Theory of Relations*, volume 118 of *Studies in logic and the foundations of mathematics*. Elsevier Science, 1986.

[Fre78] E. Freuder. Synthesizing Constraint Expressions. In *CACM*, chapter 21, pages 958–966. 1978.

[Fre82] E.C. Freuder. A Sufficient Condition for Backtrack-Free Search. In *CACM*, chapter 19, pages 24–32. 1982.

[GB65] S. W. Golomb and L. D. Baumert. Backtrack programming. *Journal of the ACM*, 12(4):516–524, 1965.

[Gea80] G. Gierz and K.H. Hofman et al. *A Compendium of Continuous Lattices*. Springer Verlag, Berlin Heidelberg New York, 1980. Chapter 0.

[Ger93a] C. Gervet. New structures of symbolic constraint objects: sets and graphs. In *WCLP'93*, Marseille, France, March 1993.

[Ger93b] C. Gervet. Sets and binary relation variables viewed as constrained objects. In *Workshop on Logic Programming with Sets*, pages 5–8, Budapest, Hungary, June 1993. In conjunction with ICLP'93.

[Ger94] C. Gervet. Conjunto : Constraint Logic Programming with Finite Set Domains. In M. Bruynooghe, editor, *ILPS'94*, pages 339–358, 1994.

[GJ79] M.R. Garey and D. S. Johnson. *Computers and intractability, A guide to the theory of NP-completeness.* Victor Klee, 1979. 124-130.

[GM84] M. Gondran and M. Minoux. *Graphs and algorithms.* Series in Discrete Mathematics. Wiley-interscience, Great Britain, 1984.

[HD86] P. Van Hentenryck and M. Dincbas. Domains in Logic Programming. In *AAAI-86*, Philadelphia,PA, 1986.

[HD87] P. Van Hentenryck and M. Dincbas. Forward checking in Logic Programming. In J. L. Lassez, editor, *Proc. of the Fourth International Conference on LP*, May 1987.

[HD91] P. Van Hentenryck and Y. Deville. Operational Semantics of Constraint Logic Programming over Finite Domains. In *Proceedings of PLILP'91*, pages 396–406, Passau, Germany, Aug. 1991.

[HDT92] P. Van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[HE80] R. M. Haralick and L.G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. In *Artificial Intelligence*, volume 14, pages 263–313. 1980.

[Hen91] P. Van Hentenryck. Constraint logic programming. *The Knowledge Engineering Review*, 6(3):151–194, 1991.

[Hib95] M. Hibti. *Décidabilité et complexité de systèmes de constraintes ensemblistes.* PhD thesis, Université de Franche-Comté, Besançon, 1995. In French.

[HJ91] N. Heintze and J. Jaffar. A Decision Procedure for a Class of Set Constraints. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in CS*, pages 300–309, July 1991.

[HLL93] M. Hibti, H. Lombardi, and B. Legeard. Deciding in HFS-Theory via Linear Integer Programming with Application to Set Unification. In *LPAR'93*, pages 170–181, St Petersbourg,Russia, July 1993.

[HLL94] M. Hibti, B. Legeard, and H. Lombardi. Decision Procedure for Constraints over Sets, Multi-sets and Sequences. Rapport de recherche LAB-TRIAP9409, L.I.B., Besançon, 1994.

[Hol92] C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *PLILP'92*, pages 260–268, 1992.

[HP92] K. L. Hoffman and M. Padberg. Solving Airline Crew-Scheduling Problems by Branch-and-Cut. Technical Report 376, George Mason and New York University, April 1992.

[HSD93] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, Implementation, and Evaluation of the Constraint Language cc(fd). Technical Report CS-93-02, Brown university, 1993.

[Hui90] S. Le Huitouze. A New Datastructure for Implementing Extensions to Prolog. In *2nd Int. Work. Programming Languages Implementation and Logic Programming, LNCS 456*, pages 136–150, 1990.

[JL87] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, 1987.

[JM87] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP system. In *Fourth ICLP conference*, pages 196–218, Melbourne, 1987. ICLP.

[JM94] J. Jaffar and M. J. Maher. Constraint Logic Programming: a Survey. In *Journal of Logic Programming*, chapter 19(20), pages 503–581. 1994.

[JP89] B. Jayaraman and D.A. Plaisted. Programming with Equations, Subsets, and Relations. In Lusk and Overbeek, editors, *Proceedings of the North American Conference*, pages 1051–1068. Logic Programming, 1989.

[KKR90] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. In *Proc. 9th ACM PODS*, pages 299–313, 1990.

[KN86] D. Kapur and P. Narendran. Np-completeness of the set unification and matching problems. In *CADE*, pages 489–495, 1986.

[Kow74] R.A. Kowalski. Predicate Logic as a Programming Language. *IFIP*, pages 569–574, 1974.

[Kup90] G. Kuper. *Logic Programming with Sets*, volume 41 of *1*, pages 44–64. Academic Press, New York and London, 1990.

[Lau78] J. L. Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10:29–127, 1978.

[Lho93] O. Lhomme. Consistency Techniques for Numeric CSPs. In *Proceedings of the 13th IJCAI conference*. IJCAI, 1993.

[LL91] B. Legeard and E. Legros. Short overview of the CLPS System. In *Proceedings of PLILP'91*, Passau, Germany, Aug. 1991. 3rd International Symposium on Programming Language Implementation and Logic Programming.

[LL92] B. Legeard and E. Legros. Test de satisfaisabilité dans le langage de programmation en logique avec contraintes ensemblistes: CLPS. In *Actes des JFPL*, pages 18–34, May 1992.

[LLLH93] B. Legeard, H. Lombardi, E. Legros, and M. Hibti. A Constraint Satisfaction Approach to Set Unification. In *13th International Conference on Artificial Intelligence, Expert Systems and Natural Language, EC2*, Avignon, France, May 24-28 1993.

[Llo87] J.W. Lloyd. *Foundations of logic programming*. Springer-Verlag, 1987.

[LR80] C.C. Lindner and A. Rosa. *Topics on Steiner Systems*, volume 7 of *Annals of Discrete Mathematics*. North Holland, 1980.

[LS76] M. Livesey and J. Siekmann. Unification of Sets and Multisets. Memo seki-76-ii, University of St. Andrews (Scotland) and Universitât Karlsruhe (Germany) Department of Computer Science, 1976.

[LS78] M. Livesey and J. Siekmann. Unification of Sets and Multisets. Internal report, Universitat Karlsruhe, 1978.

[Lue89] H. Lueneburg. *Tools and fundamental Constructions of Combinatorial Mathematics*, chapter II VIII X, pages 33–55 166–204 227–238. Wissenschaftverlag, 1989.

[LvE93] J.H.M. Lee and H. van Emden. Interval Computation as Deduction in CHIP. In *Journal of Logic Programming*, chapter vol 16. numb. 3-4, pages 255–276. Elsevier, 1993.

[Mac77] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 1977.

[McG79] J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. In *Information Sciences*, chapter vol. 19, pages 229–250. 1979.

[MF85] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25, 1985.

[MH86] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 1986.

[Mon74] U. Montanari. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. In *Information Science*, chapter 7(2), pages 95–132. 1974.

[MR91] U. Montanari and F. Rossi. Constraint relaxation may be perfect. In *Journal of Artificial Intelligence*, chapter vol 48., pages 143–170. Elsevier, 1991.

[MR93] U. Montanari and F. Rossi. *Constraint Logic Programming Selected Research*, chapter Finite Domain Constraint Solving and Constraint Logic Programming, pages 201–221. MIT Press, 1993.

[Nad88] B. Nadel. Tree search and arc consistency in constraint satisfaction algorithms. In Springer-verlag, editor, *Search in Artificial Intelligence*, pages 287–342, 1988.

[OB93] W. J. Older and F. Benhamou. Programming in CLP(BNR)*. In *PPCP'93*, 1993.

[OPL89] A. OPLOBEDU. Charme: Un Langage Industriel de Programmation par Contraintes. *Avignon 89*, 1989.

[OV90] W. Older and A. Vellino. Extending Prolog with Constraint Arithmetic on Real Intervals. In *IEEE Canadian Conference on Electrical and Computer Engineering*, 1990.

[Pad79] M. W. Padberg. Covering, Packing and Knapsack Problems. In *Annals of Discrete Mathematics*, chapter volume 4, pages 265–287. North-Holland Publishing company, 1979.

[Paw84] Z. Pawlak. Rough Classification. In *International Journal of Man-Machine Studies*, chapter Num 5 May, pages 469–483. Academic Press London, 1984.

[Paw91] Z. Pawlak. *Rough Sets: Theoretical Aspects of Reasoning about Data*. D: System theory, Knowledge engineering and Problem solving. Kluwer Academic Publishers, 1991.

[PPMK86] K. J. Perry, K. V. Palem, K. MacAloon, and G. M. Kuper. The Complexity of Logic Programming with Sets. *Computer Science*, 1986.

[PS82] C. H. Papadimitriou and K. Steiglitz. *COMBINATORIAL OPTIMIZATION: Algorithms and Complexity*. Prentice-Hall, 1982.

[Pug92] J-F. Puget. Programmation par contraintes orientée objet. In *Proceedings of Avignon*, pages 129–138, Avignon, 1992. Avignon'92.

[RM89] F. Rossi and U. Montanari. Exact Solution in Linear Time of Networks of Constraints Using Perfect Relaxation. In *International Conf. on Principles of Knowledge Representation*, pages 394–399, 1989.

[RM90] F. Rossi and U. Montanari. Constraint Relaxation as Higher Order Logic Programming. In M. Bruynooghe, editor, *META'90*, pages 82–101, Leuven, 1990. Proceedings of the 2nd workshop on meta-programming in logic.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* Discrete Mathematics. Wiley-interscience, 1986.

[SRP91] V. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundation of Concurrent Constraint Programming. In *18th Symposium on Principles of Programming Languages*, pages 333–352. ACM, 1991.

[STZ92] O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of set terms in the logic data language (LDL). *The Journal of Logic Programming*, 12(12):89–119, 1992.

[Ull66] R. Ullman. Associating Parts of Patterns. In *Information Control*, chapter 9(6), pages 583–601. 1966.

[Wal60] R. L. Walker. An Enumerative Technique for a Class of Combinatorial Problems. *Amer. Math. Soc. Proc. Symp. Appl. Math.*, 10:91–94, 1960.

[Wal75] D. L. Waltz. *The Psychology of Computer Vision*, chapter Understanding line drawings of scenes of shadows. McGraw-Hill Book Company, 1975.

[Wal89] C. Walinsky. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In *ICLP'89*, pages 181–190, 1989.

[WBP95] M. Wallace, S. Bressan, and T. Le Provost. Magic Checking: Constraint Checking for Database Query Optimisation. *Proceedings of the first workshop on Constraints and Databases and their applications, CDB'95*, 1995. To appear.